Part III

Command Reference

CHAPTER

ONE

COMMANDS

1.1 angle_coeff command

1.1.1 Syntax

```
angle_coeff N args
```

- N = numeric angle type (see asterisk form below), or type label
- args = coefficients for one or more angle types

1.1.2 Examples

```
angle_coeff 1 300.0 107.0
angle_coeff * 5.0
angle_coeff 2*10 5.0

labelmap angle 1 hydroxyl
angle_coeff hydroxyl 300.0 107.0
```

1.1.3 Description

Specify the angle force field coefficients for one or more angle types. The number and meaning of the coefficients depends on the angle style. Angle coefficients can also be set in the data file read by the *read_data* command or in a restart file.

N can be specified in one of two ways. An explicit numeric value can be used, as in the first example above. Or *N* can be a type label, which is an alphanumeric string defined by the *labelmap* command or in a section of a data file read by the *read_data* command.

For numeric values only, a wild-card asterisk can be used to set the coefficients for multiple angle types. This takes the form "*" or "*n" or "n*" or "m*n". If *N* is the number of angle types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

Note that using an *angle_coeff* command can override a previous setting for the same angle type. For example, these commands set the coeffs for all angle types, then overwrite the coeffs for just angle type 2:

```
angle_coeff * 200.0 107.0 1.2
angle_coeff 2 50.0 107.0
```

A line in a data file that specifies angle coefficients uses the exact same format as the arguments of the *angle_coeff* command in an input script, except that wild-card asterisks should not be used since coefficients for all *N* types must be listed in the file. For example, under the "Angle Coeffs" section of a data file, the line that corresponds to the first example above would be listed as

```
1 300.0 107.0
```

The *angle_style class2* is an exception to this rule, in that an additional argument is used in the input script to allow specification of the cross-term coefficients. See its doc page for details.

The list of all angle styles defined in LAMMPS is given on the *angle_style* doc page. They are also listed in more compact form on the *Commands angle* doc page.

On either of those pages, click on the style to display the formula it computes and its coefficients as specified by the associated *angle_coeff* command.

1.1.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

An angle style must be defined before any angle coefficients are set, either in the input script or in a data file.

1.1.5 Related commands

angle_style

1.1.6 Default

none

1.2 angle_style command

1.2.1 Syntax

```
angle_style style
```

- style = *none* or *zero* or *hybrid* or *amoeba* or *charmm* or *class2* or *class2/p6* or *cosine* or *cosine/buck6d* or *cosine/delta* or *cosine/periodic* or *cosine/shift* or *cosine/shift/exp* or *cosine/squared* or *cosine/squared/restricted* or *cross* or *dipole* or *fourier* or *fourier/simple* or *gaussian* or *harmonic* or *lepton* or *mm3* or *quartic* or *spica* or *table*

1.2.2 Examples

```
angle_style harmonic
angle_style charmm
angle_style hybrid harmonic cosine
```

1.2.3 Description

Set the formula(s) LAMMPS uses to compute angle interactions between triplets of atoms, which remain in force for the duration of the simulation. The list of angle triplets is read in by a *read_data* or *read_restart* command from a data or restart file.

Hybrid models where angles are computed using different angle potentials can be setup using the *hybrid* angle style.

The coefficients associated with a angle style can be specified in a data or restart file or via the *angle_coeff* command.

All angle potentials store their coefficient data in binary restart files which means angle_style and *angle_coeff* commands do not need to be re-specified in an input script that restarts a simulation. See the *read_restart* command for details on how to do this. The one exception is that angle_style *hybrid* only stores the list of sub-styles in the restart file; angle coefficients need to be re-specified.

Note

When both an angle and pair style is defined, the *special_bonds* command often needs to be used to turn off (or weight) the pairwise interaction that would otherwise exist between 3 bonded atoms.

In the formulas listed for each angle style, *theta* is the angle between the three atoms in the angle.

Here is an alphabetic list of angle styles defined in LAMMPS. Click on the style to display the formula it computes and coefficients specified by the associated *angle_coeff* command.

Click on the style to display the formula it computes, any additional arguments specified in the angle_style command, and coefficients specified by the associated *angle_coeff* command.

There are also additional accelerated pair styles included in the LAMMPS distribution for faster performance on CPUs, GPUs, and KNLs. The individual style names on the *Commands angle* page are followed by one or more of (g,i,k,o,t) to indicate which accelerated styles exist.

- *none* - turn off angle interactions
- *zero* - topology but no interactions
- *hybrid* - define multiple styles of angle interactions
- *amoeba* - AMOEBA angle
- *charmm* - CHARMM angle
- *class2* - COMPASS (class 2) angle
- *class2/p6* - COMPASS (class 2) angle expanded to 6th order
- *cosine* - angle with cosine term
- *cosine/buck6d* - same as cosine with Buckingham term between 1-3 atoms
- *cosine/delta* - angle with difference of cosines

- *cosine/periodic* - DREIDING angle
- *cosine/shift* - angle cosine with a shift
- *cosine/shift/exp* - cosine with shift and exponential term in spring constant
- *cosine/squared* - angle with cosine squared term
- *cosine/squared/restricted* - angle with restricted cosine squared term
- *cross* - cross term coupling angle and bond lengths
- *dipole* - angle that controls orientation of a point dipole
- *fourier* - angle with multiple cosine terms
- *fourier/simple* - angle with a single cosine term
- *gaussian* - multi-centered Gaussian-based angle potential
- *harmonic* - harmonic angle
- *lepton* - angle potential from evaluating a string
- *mesocnt* - piecewise harmonic and linear angle for bending-buckling of nanotubes
- *mm3* - anharmonic angle
- *quartic* - angle with cubic and quartic terms
- *spica* - harmonic angle with repulsive SPICA pair style between 1-3 atoms
- *table* - tabulated by angle

1.2.4 Restrictions

Angle styles can only be set for atom_styles that allow angles to be defined.

Most angle styles are part of the MOLECULE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info. The doc pages for individual bond potentials tell if it is part of a package.

1.2.5 Related commands

angle_coeff

1.2.6 Default

```
angle_style none
```

1.3 angle_write command

1.3.1 Syntax

```
angle_write atype N file keyword
```

- atype = angle type
- N = # of values
- file = name of file to write values to
- keyword = section name in file for this set of tabulated values

1.3.2 Examples

```
angle_write 1  500 table.txt Harmonic_1
angle_write 3 1000 table.txt Harmonic_3
```

1.3.3 Description

Added in version 8Feb2023.

Write energy and force values to a file as a function of angle for the currently defined angle potential. Force in this context means the force with respect to the angle, not the force on individual atoms. This is useful for plotting the potential function or otherwise debugging its values. The resulting file can also be used as input for use with *angle style table*.

If the file already exists, the table of values is appended to the end of the file to allow multiple tables of energy and force to be included in one file. The individual sections may be identified by the *keyword*.

The energy and force values are computed for angles ranging from 0 degrees to 180 degrees for 3 interacting atoms forming an angle type atype, using the appropriate *angle_coeff* coefficients. N evenly spaced angles are used.

For example, for N = 6, values are computed at $\theta = 0, 36, 72, 108, 144, 180$.

The file is written in the format used as input for the *angle_style table* option with *keyword* as the section name. Each line written to the file lists an index number (1-N), an angle (in degrees), an energy (in energy units), and a force (in force units per radians^2). In case a new file is created, the first line will be a comment with a "DATE:" and "UNITS:" tag with the current date and *units* settings. For subsequent invocations of the *angle_write* command for the same file, data will be appended and the current units settings will be compared to the data from the header, if present. The *angle_write* will refuse to add a table to an existing file if the units are not the same.

1.3.4 Restrictions

All force field coefficients for angle and other kinds of interactions must be set before this command can be invoked.

The table of the angle energy and force data data is created by using a separate, internally created, new LAMMPS instance with a dummy system of 3 atoms for which the angle potential energy is computed after transferring the angle style and coefficients and arranging the three atoms into the corresponding geometries. The angle force is then determined from the potential energies through numerical differentiation. As a consequence of this approach, not all angle styles are compatible. The following conditions must be met:

- The angle style must be able to write its coefficients to a data file. This condition excludes for example *angle style hybrid* and *angle style table*.
- The potential function must not have any terms that depend on geometry properties other than the angle. This condition excludes for example *angle style class2* all angle types for *angle style charmm* that have non-zero Urey-Bradley terms. Please note that the *write_angle* command has no way of checking for this condition, so the resulting tables may be bogus if the requirement is not met. It is thus recommended to make careful tests for any created tables.

1.3.5 Related commands

angle_style table, *bond_write*, *dihedral_write*, *angle_style*, *angle_coeff*

1.3.6 Default

none

1.4 atom_modify command

1.4.1 Syntax

```
atom_modify keyword values ...
```

- one or more keyword/value pairs may be appended
- keyword = *id* or *map* or *first* or *sort*

  id value = yes or no
  map value = yes or array or hash
  first value = group-ID = group whose atoms will appear first in internal atom lists
  sort values = Nfreq binsize
    Nfreq = sort atoms spatially every this many time steps
    binsize = bin size for spatial sorting (distance units)

1.4.2 Examples

```
atom_modify map yes
atom_modify map hash sort 10000 2.0
atom_modify first colloid
```

1.4.3 Description

Modify certain attributes of atoms defined and stored within LAMMPS, in addition to what is specified by the *atom_style* command. The *id* and *map* keywords must be specified before a simulation box is defined; other keywords can be specified any time.

The *id* keyword determines whether non-zero atom IDs can be assigned to each atom. If the value is *yes*, which is the default, IDs are assigned, whether you use the *create atoms* or *read_data* or *read_restart* commands to initialize atoms. If the value is *no* the IDs for all atoms are assumed to be 0.

If atom IDs are used, they must all be positive integers. They should also be unique, though LAMMPS does not check for this. Typically they should also be consecutively numbered (from 1 to Natoms), though this is not required. Molecular *atom styles* are those that store bond topology information (styles bond, angle, molecular, full). These styles require atom IDs since the IDs are used to encode the topology. Some other LAMMPS commands also require the use of atom IDs. E.g. some many-body pair styles use them to avoid double computation of the I-J interaction between two atoms.

The only reason not to use atom IDs is if you are running an atomic simulation so large that IDs cannot be uniquely assigned. For a default LAMMPS build this limit is 2^31 or about 2 billion atoms. However, even in this case, you can use 64-bit atom IDs, allowing 2^63 or about 9e18 atoms, if you build LAMMPS with the - DLAMMPS_BIGBIG switch. This is described on the *Build_settings* doc page. If atom IDs are not used, they must be specified as 0 for all atoms, e.g. in a data or restart file.

> **Note**
>
> If a *triclinic simulation box* is used, atom IDs are required, due to how neighbor lists are built.

The *map* keyword determines how atoms with specific IDs are found when required. For example, the bond (angle, etc) methods need to find the local index of an atom with a specific global ID which is a bond (angle, etc) partner. LAMMPS performs this operation efficiently by creating a "map", which is either an *array* or *hash* table, as described below.

When the *map* keyword is not specified in your input script, LAMMPS only creates a map for *atom_styles* for molecular systems which have permanent bonds (angles, etc). No map is created for atomic systems, since it is normally not needed. However some LAMMPS commands require a map, even for atomic systems, and will generate an error if one does not exist. The *map* keyword thus allows you to force the creation of a map.

Specifying a value of *yes* will create either an array-style or hash-style map, depending on the size of the system. If no atom ID is larger than 1 million, then an array-style map is used, otherwise a hash-style map is used. Specifying a value of *array* or *hash* creates an array-style or hash-style map respectively, regardless of the size of the system.

For an array-style map, each processor stores a lookup table of length N, where N is the largest atom ID in the system. This is a fast, simple method for many simulations, but requires too much memory for large simulations. For a hash-style map, a hash table is created on each processor, which finds an atom ID in constant time (independent of the global number of atom IDs). It can be slightly slower than the *array* map, but its memory cost is proportional to the number of atoms owned by a processor, i.e. N/P when N is the total number of atoms in the system and P is the number of processors.

The *first* keyword allows a *group* to be specified whose atoms will be maintained as the first atoms in each processor's list of owned atoms. This in only useful when the specified group is a small fraction of all the atoms, and there are other operations LAMMPS is performing that will be sped-up significantly by being able to loop over the smaller set of atoms. Otherwise the reordering required by this option will be a net slow-down. The *neigh_modify include* and *comm_modify group* commands are two examples of commands that require this setting to work efficiently. Several *fixes*, most notably time integration fixes like *fix nve*, also take advantage of this setting if the group they operate on is the group specified by this command. Note that specifying "all" as the group-ID effectively turns off the *first* option.

It is OK to use the *first* keyword with a group that has not yet been defined, e.g. to use the atom_modify first command at the beginning of your input script. LAMMPS does not use the group until a simulation is run.

The *sort* keyword turns on a spatial sorting or reordering of atoms within each processor's subdomain every *Nfreq* timesteps. If *Nfreq* is set to 0, then sorting is turned off. Sorting can improve cache performance and thus speed-up a LAMMPS simulation, as discussed in a paper by *(Meloni)*. Its efficacy depends on the problem size (atoms/processor), how quickly the system becomes disordered, and various other factors. As a general rule, sorting is typically more effective at speeding up simulations of liquids as opposed to solids. In tests we have done, the speed-up can range from zero to 3-4x.

Reordering is performed every *Nfreq* timesteps during a dynamics run or iterations during a minimization. More precisely, reordering occurs at the first reneighboring that occurs after the target timestep. The reordering is performed locally by each processor, using bins of the specified *binsize*. If *binsize* is set to 0.0, then a binsize equal to half the *neighbor* cutoff distance (force cutoff plus skin distance) is used, which is a reasonable value. After the atoms have been binned, they are reordered so that atoms in the same bin are adjacent to each other in the processor's 1d list of atoms.

The goal of this procedure is for atoms to put atoms close to each other in the processor's one-dimensional list of atoms that are also near to each other spatially. This can improve cache performance when pairwise interactions and neighbor lists are computed. Note that if bins are too small, there will be few atoms/bin. Likewise if bins are too large, there will be many atoms/bin. In both cases, the goal of cache locality will be undermined.

Note

Running a simulation with sorting on versus off should not change the simulation results in a statistical sense. However, a different ordering will induce round-off differences, which will lead to diverging trajectories over time when comparing two simulations. Various commands, particularly those which use random numbers (e.g. *velocity create*, and *fix langevin*), may generate (statistically identical) results which depend on the order in which atoms are processed. The order of atoms in a *dump* file will also typically change if sorting is enabled.

Note

When running simple pair-wise potentials like Lennard Jones on GPUs with the KOKKOS package, using a larger binsize (e.g. 2x larger than default) and a more frequent reordering than default (e.g. every 100 time steps) may improve performance.

1.4.4 Restrictions

The *first* and *sort* options cannot be used together. Since sorting is on by default, it will be turned off if the *first* keyword is used with a group-ID that is not "all".

1.4.5 Related commands

none

1.4.6 Default

By default, *id* is yes. By default, atomic systems (no bond topology info) do not use a map. For molecular systems (with bond topology info), the default is to use a map of either *array* or *hash* style depending on the size of the system, as explained above for the *map yes* keyword/value option. By default, a *first* group is not defined. By default, sorting is enabled with a frequency of 1000 and a binsize of 0.0, which means the neighbor cutoff will be used to set the bin size. If no neighbor cutoff is defined, sorting will be turned off.

(Meloni) Meloni, Rosati and Colombo, J Chem Phys, 126, 121102 (2007).

1.5 atom_style command

1.5.1 Syntax

```
atom_style style args
```

- style = *amoeba* or *angle* or *atomic* or *body* or *bond* or *charge* or *dielectric* or *dipole* or *dpd* or *edpd* or *electron* or *ellipsoid* or *full* or *line* or *mdpd* or *molecular* or *oxdna* or *peri* or *smd* or *sph* or *sphere* or *bpm/sphere* or *spin* or *tdpd* or *tri* or *template* or *wavepacket* or *hybrid*

  ```
  args = none for any style except the following
    body args = bstyle bstyle-args
      bstyle = style of body particles
      bstyle-args = additional arguments specific to the bstyle
                    see the Howto body doc
                    page for details
    sphere arg = 0/1 (optional) for static/dynamic particle radii
    bpm/sphere arg = 0/1 (optional) for static/dynamic particle radii
    tdpd arg = Nspecies
      Nspecies = # of chemical species
    template arg = template-ID
      template-ID = ID of molecule template specified in a separate molecule command
    hybrid args = list of one or more sub-styles, each with their args
  ```

- accelerated styles (with same args) = *angle/kk* or *atomic/kk* or *bond/kk* or *charge/kk* or *full/kk* or *molecular/kk* or *spin/kk*

1.5.2 Examples

```
atom_style atomic
atom_style bond
atom_style full
atom_style body nparticle 2 10
atom_style hybrid charge bond
atom_style hybrid charge body nparticle 2 5
atom_style spin
atom_style template myMols
atom_style hybrid template twomols charge
atom_style tdpd 2
```

1.5.3 Description

The *atom_style* command selects which per-atom attributes are associated with atoms in a LAMMPS simulation and thus stored and communicated with those atoms as well as read from and stored in data and restart files. Different models (e.g. *pair styles*) require access to specific per-atom attributes and thus require a specific atom style. For example, to compute Coulomb interactions, the atom must have a "charge" (aka "q") attribute.

A number of distinct atom styles exist that combine attributes. Some atom styles are a superset of other atom styles. Further attributes may be added to atoms either via using a hybrid style which provides a union of the attributes of the sub-styles, or via the *fix property/atom* command. The *atom_style* command must be used before a simulation is setup via a *read_data*, *read_restart*, or *create_box* command.

Note

Many of the atom styles discussed here are only enabled if LAMMPS was built with a specific package, as listed below in the Restrictions section.

Once a style is selected and the simulation box defined, it cannot be changed but only augmented with the *fix property/atom* command. So one should select an atom style general enough to encompass all attributes required. E.g. with atom style *bond*, it is not possible to define angles and use angle styles.

It is OK to use a style more general than needed, though it may be slightly inefficient because it will allocate and communicate additional unused data.

1.5.4 Atom style attributes

The atom style *atomic* has the minimum subset of per-atom attributes and is also the default setting. It encompasses the following per-atom attributes (name of the vector or array in the *Atom class* is given in parenthesis): atom-ID (tag), type (type), position (x), velocities (v), forces (f), image flags (image), group membership (mask). Since all atom styles are a superset of atom style *atomic*, they all include these attributes.

This table lists all the available atom styles, which attributes they provide, which *package* is required to use them, and what the typical applications are that use them. See the *read_data*, *create_atoms*, and *set* commands for details on how to set these various quantities. More information about many of the styles is provided in the Additional Information section below.

Atom style	Attributes	Required package	Applications
amoeba	*full* + "1-5 special neighbor data"	*AMOEBA*	AMOEBA/HIPPO force fields
angle	*bond* + "angle data"	*MOLECULE*	bead-spring polymers with stiffness
atomic	tag, type, x, v, f, image, mask		atomic liquids, solids, metals
body	*atomic* + radius, rmass, angmom, torque, body	*BODY*	arbitrary bodies, see *body howto*
bond	*atomic* + molecule, nspecial, special + "bond data"	*MOLECULE*	bead-spring polymers
bpm/sphere	*bond* + radius, rmass, omega, torque, quat	*BPM*	granular bonded particle models, see *BPM howto*
charge	*atomic* + q		atomic systems with charges
dielectric	*full* + mu, area, ed, em, epsilon, curvature, q_scaled	*DIELECTRIC*	systems with surface polarization
dipole	*charge* + mu	*DIPOLE*	atomic systems with charges and point dipoles
dpd	*atomic* + rho + "reactive DPD data"	*DPD-REACT*	reactive DPD
edpd	*atomic* + "eDPD data"	*DPD-MESO*	Energy conservative DPD (eDPD)
electron	*charge* + espin, eradius, ervel, erforce	*EFF*	Electron force field systems
ellipsoid	*atomic* + rmass, angmom, torque, ellipsoid		aspherical particles
full	*molecular* + q	*MOLECULE*	molecular force fields
line	*atomic* + molecule, radius, rmass, omega, torque, line		2-d rigid body particles
mdpd	*atomic* + rho, drho, vest	*DPD-MESO*	Many-body DPD (mDPD)
molecular	*angle* + "dihedral and improper data"	*MOLECULE*	apolar and uncharged molecules
oxdna	*atomic* + id5p	*CG-DNA*	coarse-grained DNA and RNA models
peri	*atomic* + rmass, vfrac, s0, x0	*PERI*	mesoscopic Peridynamics models
smd	*atomic* + molecule, radius, rmass + "smd data"	*MACHDYN*	Smooth Mach Dynamics models
rheo	*atomic* + rho, status	*RHEO*	solid and fluid RHEO particles
rheo/thermal	*atomic* + rho, status, energy, temperature	*RHEO*	RHEO particles with temperature
sph	*atomic* + "sph data"	*SPH*	Smoothed particle hydrodynamics models
sphere	*atomic* + radius, rmass, omega, torque		finite size spherical particles, e.g. granular models
spin	*atomic* + "magnetic moment data"	*SPIN*	magnetic particles
tdpd	*atomic* + cc, cc_flux, vest	*DPD-MESO*	Transport DPD (tDPD)
template	*atomic* + molecule, molindex, molatom	*MOLECULE*	molecular systems where attributes are taken from *molecule files*
tri	*sphere* + molecule, angmom, tri		3-d triangulated rigid body LJ particles
wavepacket	*charge* + "wavepacket data"	*AWPMD*	Antisymmetrized wave packet MD

Note

It is possible to add some attributes, such as a molecule ID and charge, to atom styles that do not have them built in using the *fix property/atom* command. This command also allows new custom-named attributes consisting of

extra integer or floating-point values or vectors to be added to atoms. See the *fix property/atom* page for examples of cases where this is useful and details on how to initialize, access, and output these custom values.

1.5.5 Particle size and mass

All of the atom styles define point particles unless they (1) define finite-size spherical particles via the *radius* attribute, or (2) define finite-size aspherical particles (e.g. the *body*, *ellipsoid*, *line*, and *tri* styles). Most of these styles can also be used with mixtures of point and finite-size particles.

Note that the *radius* property may need to be provided as a *diameter* (e.g. in *molecule files* or *data files*). See the *Howto spherical* page for an overview of using finite-size spherical and aspherical particle models with LAMMPS.

Unless an atom style defines the per-atom *rmass* attribute, particle masses are defined on a per-type basis, using the *mass* command. This means each particle's mass is indexed by its atom *type*.

A few styles define the per-atom *rmass* attribute which can also be added using the *fix property/atom* command. In this case each particle stores its own mass. Atom styles that have a per-atom rmass may define it indirectly through setting particle diameter and density on a per-particle basis. If both per-type mass and per-atom *rmass* are defined (e.g. in a hybrid style), the per-atom mass will take precedence in any operation which which works with both flavors of mass.

1.5.6 Additional information about specific atom styles

For the *body* style, the particles are arbitrary bodies with internal attributes defined by the "style" of the bodies, which is specified by the *bstyle* argument. Body particles can represent complex entities, such as surface meshes of discrete points, collections of sub-particles, deformable objects, etc.

The *Howto body* page describes the body styles LAMMPS currently supports, and provides more details as to the kind of body particles they represent. For all styles, each body particle stores moments of inertia and a quaternion 4-vector, so that its orientation and position can be time integrated due to forces and torques.

Note that there may be additional arguments required along with the *bstyle* specification, in the atom_style body command. These arguments are described on the *Howto body* doc page.

For the *dielectric* style, each particle can be either a physical particle (e.g. an ion), or an interface particle representing a boundary element between two regions of different dielectric constant. For interface particles, in addition to the properties associated with atom_style full, each particle also should be assigned a unit dipole vector (mu) representing the direction of the induced dipole moment at each interface particle, an area (area/patch), the difference and mean of the dielectric constants of two sides of the interface along the direction of the normal vector (ed and em), the local dielectric constant at the boundary element (epsilon), and a mean local curvature (curv). Physical particles must be assigned these values, as well, but only their local dielectric constants will be used; see documentation for associated *pair styles* and *fixes*. The distinction between the physical and interface particles is only meaningful when *fix polarize* commands are applied to the interface particles. This style is part of the DIELECTRIC package.

For the *dipole* style, a point dipole vector mu is defined for each point particle. Note that if you wish the particles to be finite-size spheres as in a Stockmayer potential for a dipolar fluid, so that the particles can rotate due to dipole-dipole interactions, then you need to use the command *atom_style hybrid sphere dipole*, which will assign both a diameter and dipole moment to each particle. This also requires using an integrator with a "/sphere" suffix like *fix nve/sphere* or *fix nvt/sphere* and the "update dipole" or "update dlm" parameters to the fix commands.

The *dpd* style is for reactive dissipative particle dynamics (DPD) particles. Note that it is part of the DPD-REACT package, and is not required for use with the *pair_style dpd or dpd/stat* commands, which only require the attributes from

atom_style *atomic*. Atom_style *dpd* extends DPD particle properties with internal temperature (dpdTheta), internal conductive energy (uCond), internal mechanical energy (uMech), and internal chemical energy (uChem).

The *edpd* style is for energy-conserving dissipative particle dynamics (eDPD) particles which store a temperature (edpd_temp), and heat capacity (edpd_cv).

For the *electron* style, the particles representing electrons are 3d Gaussians with a specified position and bandwidth or uncertainty in position, which is represented by the eradius = electron size.

For the *ellipsoid* style, particles can be ellipsoids which each stores a shape vector with the 3 diameters of the ellipsoid and a quaternion 4-vector with its orientation. Each particle stores a flag in the ellipsoid vector which indicates whether it is an ellipsoid (1) or a point particle (0).

For the *line* style, particles can be are idealized line segments which store a per-particle mass and length and orientation (i.e. the end points of the line segment). Each particle stores a flag in the line vector which indicates whether it is a line segment (1) or a point particle (0).

The *mdpd* style is for many-body dissipative particle dynamics (mDPD) particles which store a density (rho) for considering density-dependent many-body interactions.

The *oxdna* style is for coarse-grained nucleotides and stores the 3'-to-5' polarity of the nucleotide strand, which is set through the bond topology in the data file. The first (second) atom in a bond definition is understood to point towards the 3'-end (5'-end) of the strand.

For the *peri* style, the particles are spherical and each stores a per-particle mass and volume.

The *smd* style is for Smooth Particle Mach dynamics. Both fluids and solids can be modeled. Particles store the mass and volume of an integration point, a kernel diameter used for calculating the field variables (e.g. stress and deformation) and a contact radius for calculating repulsive forces which prevent individual physical bodies from penetrating each other.

The *sph* style is for smoothed particle hydrodynamics (SPH) particles which store a density (rho), energy (esph), and heat capacity (cv).

For the *spin* style, a magnetic spin is associated with each atom. Those spins have a norm (their magnetic moment) and a direction.

The *tdpd* style is for transport dissipative particle dynamics (tDPD) particles which store a set of chemical concentration. An integer "cc_species" is required to specify the number of chemical species involved in a tDPD system.

The *wavepacket* style is similar to the *electron* style, but the electrons may consist of several Gaussian wave packets, summed up with coefficients cs= (cs_re,cs_im). Each of the wave packets is treated as a separate particle in LAMMPS, wave packets belonging to the same electron must have identical *etag* values.

The *sphere* and *bpm/sphere* styles allow particles to be either point particles or finite-size particles. If the *radius* attribute is > 0.0, the particle is a finite-size sphere. If the diameter = 0.0, it is a point particle. Note that by using the *disc* keyword with the *fix nve/sphere*, *fix nvt/sphere*, *fix nph/sphere*, *fix npt/sphere* commands for the *sphere* style, spheres can be effectively treated as 2d discs for a 2d simulation if desired. See also the *set density/disc* command. These styles also take an optional 0 or 1 argument. A value of 0 means the radius of each sphere is constant for the duration of the simulation (this is the default). A value of 1 means the radii may vary dynamically during the simulation, e.g. due to use of the *fix adapt* command.

The *template* style allows molecular topology (bonds,angles,etc) to be defined via a molecule template using the *molecule* command. The template stores one or more molecules with a single copy of the topology info (bonds,angles,etc) of each. Individual atoms only store a template index and template atom to identify which molecule and which atom-within-the-molecule they represent. Using the *template* style instead of the *bond*, *angle*, *molecular* styles can save memory for systems comprised of a large number of small molecules, all of a single type (or small number of types). See the paper by Grime and Voth, in *(Grime)*, for examples of how this can be advantageous for large-scale coarse-grained systems. The examples/template directory has a few demo inputs and examples showing the use of the *template* atom style versus *molecular*.

Note

When using the *template* style with a *molecule template* that contains multiple molecules, you should ensure the atom types, bond types, angle_types, etc in all the molecules are consistent. E.g. if one molecule represents H2O and another CO2, then you probably do not want each molecule file to define two atom types and a single bond type, because they will conflict with each other when a mixture system of H2O and CO2 molecules is defined, e.g. by the *read_data* command. Rather the H2O molecule should define atom types 1 and 2, and bond type 1. And the CO2 molecule should define atom types 3 and 4 (or atom types 3 and 2 if a single oxygen type is desired), and bond type 2.

For the *tri* style, particles can be planar triangles which each stores a per-particle mass and size and orientation (i.e. the corner points of the triangle). Each particle stores a flag in the tri vector which indicates whether it is a triangle (1) or a point particle (0).

Typically, simulations require only a single (non-hybrid) atom style. If some atoms in the simulation do not have all the properties defined by a particular style, use the simplest style that defines all the needed properties by any atom. For example, if some atoms in a simulation are charged, but others are not, use the *charge* style. If some atoms have bonds, but others do not, use the *bond* style.

The only scenario where the *hybrid* style is needed is if there is no single style which defines all needed properties of all atoms. For example, as mentioned above, if you want dipolar particles which will rotate due to torque, you need to use "atom_style hybrid sphere dipole". When a hybrid style is used, atoms store and communicate the union of all quantities implied by the individual styles.

When using the *hybrid* style, you cannot combine the *template* style with another molecular style that stores bond, angle, etc info on a per-atom basis.

LAMMPS can be extended with new atom styles as well as new body styles; see the corresponding manual page on *modifying & extending LAMMPS*.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

1.5.7 Restrictions

This command cannot be used after the simulation box is defined by a *read_data* or *create_box* command.

Many of the styles listed above are only enabled if LAMMPS was built with a specific package, as listed below. See the *Build package* page for more info. The table above lists which package is required for individual atom styles.

1.5.8 Related commands

read_data, *pair_style*, *fix property/atom*, *set*

1.5.9 Default

The default atom style is *atomic*. If atom_style *sphere* or *bpm/sphere* is used, its default argument is 0.

(Grime) Grime and Voth, to appear in J Chem Theory & Computation (2014).

1.6 balance command

1.6.1 Syntax

```
balance thresh style args ... keyword args ...
```

- thresh = imbalance threshold that must be exceeded to perform a re-balance
- one style/arg pair can be used (or multiple for *x*,*y*,*z*)
- style = *x* or *y* or *z* or *shift* or *rcb*

  x args = uniform or Px-1 numbers between 0 and 1
    uniform = evenly spaced cuts between processors in x dimension
    numbers = Px-1 ascending values between 0 and 1, Px - # of processors in x dimension
    x can be specified together with y or z
  y args = uniform or Py-1 numbers between 0 and 1
    uniform = evenly spaced cuts between processors in y dimension
    numbers = Py-1 ascending values between 0 and 1, Py - # of processors in y dimension
    y can be specified together with x or z
  z args = uniform or Pz-1 numbers between 0 and 1
    uniform = evenly spaced cuts between processors in z dimension
    numbers = Pz-1 ascending values between 0 and 1, Pz - # of processors in z dimension
    z can be specified together with x or y
  shift args = dimstr Niter stopthresh
    dimstr = sequence of letters containing "x" or "y" or "z", each not more than once
    Niter = # of times to iterate within each dimension of dimstr sequence
    stopthresh = stop balancing when this imbalance threshold is reached
  rcb args = none

- zero or more keyword/arg pairs may be appended
- keyword = *weight* or *out*

weight style args = use weighted particle counts for the balancing
  style = group or neigh or time or var or store
    group args = Ngroup group1 weight1 group2 weight2 ...
      Ngroup = number of groups with assigned weights
      group1, group2, ... = group IDs
      weight1, weight2, ... = corresponding weight factors
    neigh factor = compute weight based on number of neighbors
      factor = scaling factor (> 0)
    time factor = compute weight based on time spend computing
      factor = scaling factor (> 0)
    var name = take weight from atom-style variable
      name = name of the atom-style variable
    store name = store weight in custom atom property defined by fix property/atom command
      name = atom property name (without d_ prefix)
sort arg = no or yes
out arg = filename
  filename = write each processor's subdomain to a file

1.6.2 Examples

```
balance 0.9 x uniform y 0.4 0.5 0.6
balance 1.2 shift xz 5 1.1
balance 1.0 shift xz 5 1.1
balance 1.1 rcb
balance 1.0 shift x 10 1.1 weight group 2 fast 0.5 slow 2.0
balance 1.0 shift x 10 1.1 weight time 0.8 weight neigh 0.5 weight store balance
balance 1.0 shift x 20 1.0 out tmp.balance
```

1.6.3 Description

This command adjusts the size and shape of processor subdomains within the simulation box, to attempt to balance the number of atoms or particles and thus indirectly the computational cost (load) more evenly across processors. The load balancing is "static" in the sense that this command performs the balancing once, before or between simulations. The processor subdomains will then remain static during the subsequent run. To perform "dynamic" balancing, see the *fix balance* command, which can adjust processor subdomain sizes and shapes on-the-fly during a *run*.

Load-balancing is typically most useful if the particles in the simulation box have a spatially-varying density distribution or when the computational cost varies significantly between different particles. E.g. a model of a vapor/liquid interface, or a solid with an irregular-shaped geometry containing void regions, or *hybrid pair style simulations* which combine pair styles with different computational cost. In these cases, the LAMMPS default of dividing the simulation box volume into a regular-spaced grid of 3d bricks, with one equal-volume subdomain per processor, may assign numbers of particles per processor in a way that the computational effort varies significantly. This can lead to poor performance when the simulation is run in parallel.

The balancing can be performed with or without per-particle weighting. With no weighting, the balancing attempts to assign an equal number of particles to each processor. With weighting, the balancing attempts to assign an equal aggregate computational weight to each processor, which typically induces a different number of atoms assigned to each processor. Details on the various weighting options and examples for how they can be used are *given below*.

Note that the *processors* command allows some control over how the box volume is split across processors. Specifically, for a Px by Py by Pz grid of processors, it allows choice of Px, Py, and Pz, subject to the constraint that Px * Py * Pz = P, the total number of processors. This is sufficient to achieve good load-balance for some problems on some processor counts. However, all the processor subdomains will still have the same shape and same volume.

The requested load-balancing operation is only performed if the current "imbalance factor" in particles owned by each processor exceeds the specified *thresh* parameter. The imbalance factor is defined as the maximum number of particles (or weight) owned by any processor, divided by the average number of particles (or weight) per processor. Thus an imbalance factor of 1.0 is perfect balance.

As an example, for 10000 particles running on 10 processors, if the most heavily loaded processor has 1200 particles, then the factor is 1.2, meaning there is a 20% imbalance. Note that a re-balance can be forced even if the current balance is perfect (1.0) be specifying a *thresh* < 1.0.

Note

Balancing is performed even if the imbalance factor does not exceed the *thresh* parameter if a "grid" style is specified when the current partitioning is "tiled". The meaning of "grid" vs "tiled" is explained below. This is to allow forcing of the partitioning to "grid" so that the *comm_style brick* command can then be used to replace a current *comm_style tiled* setting.

When the balance command completes, it prints statistics about the result, including the change in the imbalance factor and the change in the maximum number of particles on any processor. For "grid" methods (defined below) that create a logical 3d grid of processors, the positions of all cutting planes in each of the 3 dimensions (as fractions of the box length) are also printed.

Note

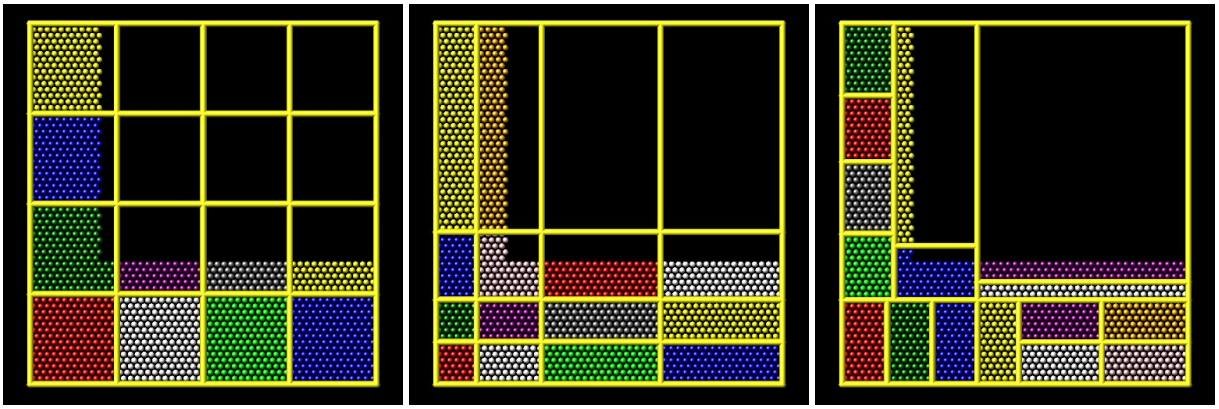
This command attempts to minimize the imbalance factor, as defined above. But depending on the method a perfect balance (1.0) may not be achieved. For example, "grid" methods (defined below) that create a logical 3d grid cannot achieve perfect balance for many irregular distributions of particles. Likewise, if a portion of the system is a perfect lattice, e.g. the initial system is generated by the *create_atoms* command, then "grid" methods may be unable to achieve exact balance. This is because entire lattice planes will be owned or not owned by a single processor.

Note

The imbalance factor is also an estimate of the maximum speed-up you can hope to achieve by running a perfectly balanced simulation versus an imbalanced one. In the example above, the 10000 particle simulation could run up to 20% faster if it were perfectly balanced, versus when imbalanced. However, computational cost is not strictly proportional to particle count, and changing the relative size and shape of processor subdomains may lead to additional computational and communication overheads, e.g. in the PPPM solver used via the *kspace_style* command. Thus you should benchmark the run times of a simulation before and after balancing.

The method used to perform a load balance is specified by one of the listed styles (or more in the case of *x*,*y*,*z*), which are described in detail below. There are 2 kinds of styles.

The *x*, *y*, *z*, and *shift* styles are "grid" methods which produce a logical 3d grid of processors. They operate by changing the cutting planes (or lines) between processors in 3d (or 2d), to adjust the volume (area in 2d) assigned to each processor, as in the following 2d diagram where processor subdomains are shown and particles are colored by the processor that owns them.



The leftmost diagram is the default partitioning of the simulation box across processors (one sub-box for each of 16 processors); the middle diagram is after a "grid" method has been applied. The *rcb* style is a "tiling" method which does not produce a logical 3d grid of processors. Rather it tiles the simulation domain with rectangular sub-boxes of varying size and shape in an irregular fashion so as to have equal numbers of particles (or weight) in each sub-box, as in the rightmost diagram above.

The "grid" methods can be used with either of the *comm_style* command options, *brick* or *tiled*. The "tiling" methods can only be used with *comm_style tiled*. Note that it can be useful to use a "grid" method with *comm_style tiled* to return the domain partitioning to a logical 3d grid of processors so that "comm_style brick" can afterwords be specified for subsequent *run* commands.

When a "grid" method is specified, the current domain partitioning can be either a logical 3d grid or a tiled partitioning. In the former case, the current logical 3d grid is used as a starting point and changes are made to improve the imbalance factor. In the latter case, the tiled partitioning is discarded and a logical 3d grid is created with uniform spacing in all dimensions. This becomes the starting point for the balancing operation.

When a "tiling" method is specified, the current domain partitioning ("grid" or "tiled") is ignored, and a new partitioning is computed from scratch.

The *x*, *y*, and *z* styles invoke a "grid" method for balancing, as described above. Note that any or all of these 3 styles can be specified together, one after the other, but they cannot be used with any other style. This style adjusts the position of cutting planes between processor subdomains in specific dimensions. Only the specified dimensions are altered.

The *uniform* argument spaces the planes evenly, as in the left diagrams above. The *numeric* argument requires listing Ps-1 numbers that specify the position of the cutting planes. This requires knowing Ps = Px or Py or Pz = the number of processors assigned by LAMMPS to the relevant dimension. This assignment is made (and the Px, Py, Pz values printed out) when the simulation box is created by the "create_box" or "read_data" or "read_restart" command and is influenced by the settings of the *processors* command.

Each of the numeric values must be between 0 and 1, and they must be listed in ascending order. They represent the fractional position of the cutting place. The left (or lower) edge of the box is 0.0, and the right (or upper) edge is 1.0. Neither of these values is specified. Only the interior Ps-1 positions are specified. Thus is there are 2 processors in the x dimension, you specify a single value such as 0.75, which would make the left processor's subdomain 3x larger than the right processor's subdomain.

The *shift* style invokes a "grid" method for balancing, as described above. It changes the positions of cutting planes between processors in an iterative fashion, seeking to reduce the imbalance factor, similar to how the *fix balance shift* command operates.

The *dimstr* argument is a string of characters, each of which must be an "x" or "y" or "z". Each character can appear zero or one time, since there is no advantage to balancing on a dimension more than once. You should normally only list dimensions where you expect there to be a density variation in the particles.

Balancing proceeds by adjusting the cutting planes in each of the dimensions listed in *dimstr*, one dimension at a time. For a single dimension, the balancing operation (described below) is iterated on up to *Niter* times. After each dimension finishes, the imbalance factor is re-computed, and the balancing operation halts if the *stopthresh* criterion is met.

A re-balance operation in a single dimension is performed using a recursive multisectioning algorithm, where the position of each cutting plane (line in 2d) in the dimension is adjusted independently. This is similar to a recursive bisectioning for a single value, except that the bounds used for each bisectioning take advantage of information from neighboring cuts if possible. At each iteration, the count of particles on either side of each plane is tallied. If the counts do not match the target value for the plane, the position of the cut is adjusted to be halfway between a low and high bound. The low and high bounds are adjusted on each iteration, using new count information, so that they become closer together over time. Thus as the recursion progresses, the count of particles on either side of the plane gets closer to the target value.

After the balanced plane positions are determined, if any pair of adjacent planes are closer together than the neighbor skin distance (as specified by the *neigh_modify* command), then the plane positions are shifted to separate them by at least this amount. This is to prevent particles being lost when dynamics are run with processor subdomains that are too narrow in one or more dimensions.

Once the re-balancing is complete and final processor subdomains assigned, particles are migrated to their new owning processor, and the balance procedure ends.

Note

At each re-balance operation, the bisectioning for each cutting plane (line in 2d) typically starts with low and high bounds separated by the extent of a processor's subdomain in one dimension. The size of this bracketing region shrinks by 1/2 every iteration. Thus if *Niter* is specified as 10, the cutting plane will typically be positioned to 1 part in 1000 accuracy (relative to the perfect target position). For *Niter* = 20, it will be accurate to 1 part in a million. Thus there is no need to set *Niter* to a large value. LAMMPS will check if the threshold accuracy is reached (in a dimension) is less iterations than *Niter* and exit early. However, *Niter* should also not be set too small, since it will take roughly the same number of iterations to converge even if the cutting plane is initially close to the target value.

The *rcb* style invokes a "tiled" method for balancing, as described above. It performs a recursive coordinate bisectioning (RCB) of the simulation domain. The basic idea is as follows.

The simulation domain is cut into 2 boxes by an axis-aligned cut in one of the dimensions, leaving one new sub-box on either side of the cut. Which dimension is chosen for the cut depends on the particle (weight) distribution within the parent box. Normally the longest dimension of the box is cut, but if all (or most) of the particles are at one end of the box, a cut may be performed in another dimension to induce sub-boxes that are more cube-ish (3d) or square-ish (2d) in shape.

After the cut is made, all the processors are also partitioned into 2 groups, half assigned to the box on the lower side of the cut, and half to the box on the upper side. (If the processor count is odd, one side gets an extra processor.) The cut is positioned so that the number of (weighted) particles in the lower box is exactly the number that the processors assigned to that box should own for load balance to be perfect. This also makes load balance for the upper box perfect. The positioning of the cut is done iteratively, by a bisectioning method (median search). Note that counting particles on either side of the cut requires communication between all processors at each iteration.

That is the procedure for the first cut. Subsequent cuts are made recursively, in exactly the same manner. The subset of processors assigned to each box make a new cut in one dimension of that box, splitting the box, the subset of processors, and the particles in the box in two. The recursion continues until every processor is assigned a sub-box of the entire simulation domain, and owns the (weighted) particles in that sub-box.

This subsection describes how to perform weighted load balancing using the *weight* keyword.

By default, all particles have a weight of 1.0, which means each particle is assumed to require the same amount of computation during a timestep. There are, however, scenarios where this is not a good assumption. Measuring the computational cost for each particle accurately would be impractical and slow down the computation. Instead the *weight* keyword implements several ways to influence the per-particle weights empirically by properties readily available or using the user's knowledge of the system. Note that the absolute value of the weights are not important; only their relative ratios affect which particle is assigned to which processor. A particle with a weight of 2.5 is assumed to require 5x more computational than a particle with a weight of 0.5. For all the options below the weight assigned to a particle must be a positive value; an error will be be generated if a weight is <= 0.0.

Below is a list of possible weight options with a short description of their usage and some example scenarios where they might be applicable. It is possible to apply multiple weight flags and the weightings they induce will be combined through multiplication. Most of the time, however, it is sufficient to use just one method.

The *group* weight style assigns weight factors to specified *groups* of particles. The *group* style keyword is followed by the number of groups, then pairs of group IDs and the corresponding weight factor. If a particle belongs to none of the specified groups, its weight is not changed. If it belongs to multiple groups, its weight is the product of the weight factors.

This weight style is useful in combination with pair style *hybrid*, e.g. when combining a more costly many-body potential with a fast pairwise potential. It is also useful when using *run_style respa* where some portions of the system have many bonded interactions and others none. It assumes that the computational cost for each group remains constant over time. This is a purely empirical weighting, so a series test runs to tune the assigned weight factors for optimal performance is recommended.

The *neigh* weight style assigns the same weight to each particle owned by a processor based on the total count of neighbors in the neighbor list owned by that processor. The motivation is that more neighbors means a higher computational cost. The style does not use neighbors per atom to assign a unique weight to each atom, because that value can vary depending on how the neighbor list is built.

The *factor* setting is applied as an overall scale factor to the *neigh* weights which allows adjustment of their impact on the balancing operation. The specified *factor* value must be positive. A value > 1.0 will increase the weights so that the ratio of max weight to min weight increases by *factor*. A value < 1.0 will decrease the weights so that the ratio of max weight to min weight decreases by *factor*. In both cases the intermediate weight values increase/decrease proportionally as well. A value = 1.0 has no effect on the *neigh* weights. As a rule of thumb, we have found a *factor* of about 0.8 often results in the best performance, since the number of neighbors is likely to overestimate the ideal weight.

This weight style is useful for systems where there are different cutoffs used for different pairs of interactions, or the density fluctuates, or a large number of particles are in the vicinity of a wall, or a combination of these effects. If a simulation uses multiple neighbor lists, this weight style will use the first suitable neighbor list it finds. It will not request or compute a new list. A warning will be issued if there is no suitable neighbor list available or if it is not current, e.g. if the balance command is used before a *run* or *minimize* command is used, in which case the neighbor list may not yet have been built. In this case no weights are computed. Inserting a *run 0 post no* command before issuing the *balance* command, may be a workaround for this case, as it will induce the neighbor list to be built.

The *time* weight style uses *timer data* to estimate weights. It assigns the same weight to each particle owned by a processor based on the total computational time spent by that processor. See details below on what time window is used. It uses the same timing information as is used for the *MPI task timing breakdown*, namely, for sections *Pair*, *Bond*, *Kspace*, and *Neigh*. The time spent in those portions of the timestep are measured for each MPI rank, summed, then divided by the number of particles owned by that processor. I.e. the weight is an effective CPU time/particle averaged over the particles on that processor.

The *factor* setting is applied as an overall scale factor to the *time* weights which allows adjustment of their impact on the balancing operation. The specified *factor* value must be positive. A value > 1.0 will increase the weights so that the ratio of max weight to min weight increases by *factor*. A value < 1.0 will decrease the weights so that the ratio of max weight to min weight decreases by *factor*. In both cases the intermediate weight values increase/decrease proportionally as well. A value = 1.0 has no effect on the *time* weights. As a rule of thumb, effective values to use are typically between 0.5 and 1.2. Note that the timer quantities mentioned above can be affected by communication which

occurs in the middle of the operations, e.g. pair styles with intermediate exchange of data witin the force computation, and likewise for KSpace solves.

When using the *time* weight style with the *balance* command, the timing data is taken from the preceding run command, i.e. the timings are for the entire previous run. For the *fix balance* command the timing data is for only the timesteps since the last balancing operation was performed. If timing information for the required sections is not available, e.g. at the beginning of a run, or when the *timer* command is set to either *loop* or *off*, a warning is issued. In this case no weights are computed.

Note

The *time* weight style is the most generic option, and should be tried first, unless the *group* style is easily applicable. However, since the computed cost function is averaged over all particles on a processor, the weights may not be highly accurate. This style can also be effective as a secondary weight in combination with either *group* or *neigh* to offset some of inaccuracies in either of those heuristics.

The *var* weight style assigns per-particle weights by evaluating an *atom-style variable* specified by *name*. This is provided as a more flexible alternative to the *group* weight style, allowing definition of a more complex heuristics based on information (global and per atom) available inside of LAMMPS. For example, atom-style variables can reference the position of a particle, its velocity, the volume of its Voronoi cell, etc.

The *store* weight style does not compute a weight factor. Instead it stores the current accumulated weights in a custom per-atom vector specified by *name*. This must be a vector defined as *d_name* via the *fix property/atom* command. This means the values in the vector can be read as part of a data file with the *read_data* command or specified with the *set* command. These weights can also be output in a *dump* file, so this is a way to examine, debug, or visualize the per-particle weights used during the load-balancing operation.

Note that the name of the custom per-atom vector is specified just as *name*, not as *d_name* as it is for other commands that use different kinds of custom atom vectors or arrays as arguments.

The *sort* keyword determines whether the communication of per-atom data to other processors during load-balancing will be random or deterministic. Random is generally faster; deterministic will ensure the new ordering of atoms on each processor is the same each time the same simulation is run. This can be useful for debugging purposes. Since the balance command is a one-time operation, the default is *yes* to perform sorting.

The *out* keyword writes a text file to the specified *filename* with the results of the balancing operation. The file contains the bounds of the subdomain for each processor after the balancing operation completes. The format of the file is compatible with the Pizza.py *mdump* tool which has support for manipulating and visualizing mesh files. An example is shown here for a balancing by 4 processors for a 2d problem:

```
ITEM: TIMESTEP
0
ITEM: NUMBER OF NODES
16
ITEM: BOX BOUNDS
0 10
0 10
0 10
ITEM: NODES
1 1 0 0 0
2 1 5 0 0
3 1 5 5 0
4 1 0 5 0
```

(continues on next page)

(continued from previous page)

```
5 1 5 0 0
6 1 10 0 0
7 1 10 5 0
8 1 5 5 0
9 1 0 5 0
10 1 5 5 0
11 1 5 10 0
12 1 10 5 0
13 1 5 5 0
14 1 10 5 0
15 1 10 10 0
16 1 5 10 0
ITEM: TIMESTEP
0
ITEM: NUMBER OF SQUARES
4
ITEM: SQUARES
1 1 1 2 3 4
2 1 5 6 7 8
3 1 9 10 11 12
4 1 13 14 15 16
```

The coordinates of all the vertices are listed in the NODES section, 5 per processor. Note that the 4 subdomains share vertices, so there will be duplicate nodes in the list.

The "SQUARES" section lists the node IDs of the 4 vertices in a rectangle for each processor (1 to 4).

For a 3d problem, the syntax is similar with 8 vertices listed for each processor, instead of 4, and "SQUARES" replaced by "CUBES".

1.6.4 Restrictions

For 2d simulations, the *z* style cannot be used. Nor can a "z" appear in *dimstr* for the *shift* style.

Balancing through recursive bisectioning (*rcb* style) requires *comm_style tiled*

1.6.5 Related commands

group, *processors*, *fix balance*, *comm_style*

1.6.6 Default

The default setting is sort = yes.

1.7 bond_coeff command

1.7.1 Syntax

```
bond_coeff N args
```

- N = numeric bond type (see asterisk form below), or type label
- args = coefficients for one or more bond types

1.7.2 Examples

```
bond_coeff 5 80.0 1.2
bond_coeff * 30.0 1.5 1.0 1.0
bond_coeff 1*4 30.0 1.5 1.0 1.0
bond_coeff 1 harmonic 200.0 1.0 # (for bond_style hybrid)

labelmap bond 5 carbonyl
bond_coeff carbonyl 80.0 1.2
```

1.7.3 Description

Specify the bond force field coefficients for one or more bond types. The number and meaning of the coefficients depends on the bond style. Bond coefficients can also be set in the data file read by the *read_data* command or in a restart file.

N can be specified in one of several ways. An explicit numeric value can be used, as in the first example above. Or *N* can be a type label, which is an alphanumeric string defined by the *labelmap* command or in a section of a data file read by the *read_data* command.

For numeric values only, a wild-card asterisk can be used to set the coefficients for multiple bond types. This takes the form "*" or "*n" or "n*" or "m*n". If *N* is the number of bond types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

Note that using a bond_coeff command can override a previous setting for the same bond type. For example, these commands set the coeffs for all bond types, then overwrite the coeffs for just bond type 2:

```
bond_coeff * 100.0 1.2
bond_coeff 2 200.0 1.2
```

A line in a data file that specifies bond coefficients uses the exact same format as the arguments of the bond_coeff command in an input script, except that wild-card asterisks should not be used since coefficients for all *N* types must be listed in the file. For example, under the "Bond Coeffs" section of a data file, the line that corresponds to the first example above would be listed as

```
5 80.0 1.2
```

The list of all bond styles defined in LAMMPS is given on the *bond_style* doc page. They are also listed in more compact form on the *Commands bond* doc page.

On either of those pages, click on the style to display the formula it computes and its coefficients as specified by the associated bond_coeff command.

1.7.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

A bond style must be defined before any bond coefficients are set, either in the input script or in a data file.

1.7.5 Related commands

bond_style

1.7.6 Default

none

1.8 bond_style command

1.8.1 Syntax

```
bond_style style args
```

- style = *none* or *zero* or *hybrid* or *bpm/rotational* or *bpm/spring* or *class2* or *fene* or *fene/expand* or *fene/nm* or *gaussian* or *gromos* or *harmonic* or *harmonic/restrain harmonic/shift* or *harmonic/shift/cut* or *lepton* or *morse* or *nonlinear* or *oxdna/fene* or *oxdena2/fene* or *oxrna2/fene* or *quartic* or *special* or *table*
- args = none for any style except *hybrid*
  - *hybrid* args = list of one or more styles

1.8.2 Examples

```
bond_style harmonic
bond_style fene
bond_style hybrid harmonic fene
```

1.8.3 Description

Set the formula(s) LAMMPS uses to compute bond interactions between pairs of atoms. In LAMMPS, a bond differs from a pairwise interaction, which are set via the *pair_style* command. Bonds are defined between specified pairs of atoms and remain in force for the duration of the simulation (unless new bonds are created or existing bonds break, which is possible in some fixes and bond potentials). The list of bonded atoms is read in by a *read_data* or *read_restart* command from a data or restart file. By contrast, pair potentials are typically defined between all pairs of atoms within a cutoff distance and the set of active interactions changes over time.

Hybrid models where bonds are computed using different bond potentials can be setup using the *hybrid* bond style.

The coefficients associated with a bond style can be specified in a data or restart file or via the *bond_coeff* command.

All bond potentials store their coefficient data in binary restart files which means bond_style and *bond_coeff* commands do not need to be re-specified in an input script that restarts a simulation. See the *read_restart* command for details on how to do this. The one exception is that bond_style *hybrid* only stores the list of sub-styles in the restart file; bond coefficients need to be re-specified.

Note

When both a bond and pair style is defined, the *special_bonds* command often needs to be used to turn off (or weight) the pairwise interaction that would otherwise exist between two bonded atoms.

In the formulas listed for each bond style, r is the distance between the two atoms in the bond.

Here is an alphabetic list of bond styles defined in LAMMPS. Click on the style to display the formula it computes and coefficients specified by the associated *bond_coeff* command.

Click on the style to display the formula it computes, any additional arguments specified in the bond_style command, and coefficients specified by the associated *bond_coeff* command.

There are also additional accelerated pair styles included in the LAMMPS distribution for faster performance on CPUs, GPUs, and KNLs. The individual style names on the *Commands bond* doc page are followed by one or more of (g,i,k,o,t) to indicate which accelerated styles exist.

- *none* - turn off bonded interactions
- *zero* - topology but no interactions
- *hybrid* - define multiple styles of bond interactions
- *bpm/rotational* - breakable bond with forces and torques based on deviation from reference state
- *bpm/spring* - breakable bond with forces based on deviation from reference length
- *class2* - COMPASS (class 2) bond
- *fene* - FENE (finite-extensible non-linear elastic) bond
- *fene/expand* - FENE bonds with variable size particles
- *fene/nm* - FENE bonds with a generalized Lennard-Jones potential
- *gaussian* - multicentered Gaussian-based bond potential
- *gromos* - GROMOS force field bond
- *harmonic* - harmonic bond
- *harmonic/restrain* - harmonic bond to restrain to original bond distance
- *harmonic/shift* - shifted harmonic bond
- *harmonic/shift/cut* - shifted harmonic bond with a cutoff
- *lepton* - bond potential from evaluating a string
- *mesocnt* - Harmonic bond wrapper with parameterization presets for nanotubes
- *mm3* - MM3 anharmonic bond
- *morse* - Morse bond
- *nonlinear* - nonlinear bond

- *oxdna/fene* - modified FENE bond suitable for DNA modeling
- *oxdna2/fene* - same as oxdna but used with different pair styles
- *oxrna2/fene* - modified FENE bond suitable for RNA modeling
- *quartic* - breakable quartic bond
- *rheo/shell* - shell bond for oxidation modeling in RHEO
- *special* - enable special bond exclusions for 1-5 pairs and beyond
- *table* - tabulated by bond length

1.8.4 Restrictions

Bond styles can only be set for atom styles that allow bonds to be defined.

Most bond styles are part of the MOLECULE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info. The doc pages for individual bond potentials tell if it is part of a package.

1.8.5 Related commands

bond_coeff, *delete_bonds*

1.8.6 Default

```
bond_style none
```

1.9 bond_write command

1.9.1 Syntax

```
bond_write btype N inner outer file keyword itype jtype
```

- btype = bond type
- N = # of values
- inner,outer = inner and outer bond length (distance units)
- file = name of file to write values to
- keyword = section name in file for this set of tabulated values
- itype,jtype = two atom types (optional)

1.9.2 Examples

```
bond_write 1 500 0.5 3.5 table.txt Harmonic_1
bond_write 3 1000 0.1 6.0 table.txt Morse
```

1.9.3 Description

Write energy and force values to a file as a function of distance for the currently defined *bond style* for a selected bond type. This is useful for plotting the potential function or otherwise debugging its values. The resulting file can also be used as input for use with *bond style table*.

If the file already exists, the table of values is appended to the end of the file to allow multiple tables of energy and force to be included in one file. The individual sections may be identified by the *keyword*.

The energy and force values are computed at distances from *inner* to *outer* for two interacting atoms forming a bond of type *btype*, using the appropriate *bond_coeff* coefficients. N evenly spaced distances are used.

For example, for N = 7, inner = 1.0, and outer = 4.0, values are computed at r = 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0.

The file is written in the format used as input for the *bond_style table* option with *keyword* as the section name. Each line written to the file lists an index number (1-N), a distance (in distance units), an energy (in energy units), and a force (in force units). In case a new file is created, the first line will be a comment with a "DATE:" and "UNITS:" tag with the current date and *units* settings. For subsequent invocations of the *bond_write* command for the same file, data will be appended and the current units settings will be compared to the data from the header, if present. The *bond_write* command will refuse to add a table to an existing file if the units are not the same.

1.9.4 Restrictions

All force field coefficients for bond and other kinds of interactions must be set before this command can be invoked.

Due to how the bond force is computed, an inner value > 0.0 must be specified even if the potential has a finite value at r = 0.0.

1.9.5 Related commands

bond_style table, *angle_write*, *bond_style*, *bond_coeff*

1.9.6 Default

none

1.10 boundary command

1.10.1 Syntax

```
boundary x y z
```

- x,y,z = p or s or f or m, one or two letters

p is periodic
f is non-periodic and fixed
s is non-periodic and shrink-wrapped
m is non-periodic and shrink-wrapped with a minimum value

1.10.2 Examples

```
boundary p p f
boundary p fs p
boundary s f fm
```

1.10.3 Description

Set the style of boundaries for the global simulation box in each dimension. A single letter assigns the same style to both the lower and upper face of the box. Two letters assigns the first style to the lower face and the second style to the upper face. The initial size of the simulation box is set by the *read_data*, *read_restart*, or *create_box* commands.

The style *p* means the box is periodic, so that particles interact across the boundary, and they can exit one end of the box and re-enter the other end. A periodic dimension can change in size due to constant pressure boundary conditions or box deformation (see the *fix npt* and *fix deform* commands). The *p* style must be applied to both faces of a dimension. For 2d simulations the z dimension must be periodic (which is the default).

The styles *f*, *s*, and *m* mean the box is non-periodic, so that particles do not interact across the boundary and do not move from one side of the box to the other.

For style *f*, the position of the face is fixed. If an atom moves outside the face it will be deleted on the next timestep that reneighboring occurs. This will typically generate an error unless you have set the *thermo_modify lost* option to allow for lost atoms.

For style *s*, the position of the face is set so as to encompass the atoms in that dimension (shrink-wrapping), no matter how far they move. Note that when the difference between the current box dimensions and the shrink-wrap box dimensions is large, this can lead to lost atoms at the beginning of a run when running in parallel. This is due to the large change in the (global) box dimensions also causing significant changes in the individual subdomain sizes. If these changes are farther than the communication cutoff, atoms will be lost. This is best addressed by setting initial box dimensions to match the shrink-wrapped dimensions more closely, by using *m* style boundaries (see below).

For style *m*, shrink-wrapping occurs, but is bounded by the value specified in the data or restart file or set by the *create_box* command. For example, if the upper z face has a value of 50.0 in the data file, the face will always be positioned at 50.0 or above, even if the maximum z-extent of all the atoms becomes less than 50.0. This can be useful if you start a simulation with an empty box or if you wish to leave room on one side of the box, e.g. for atoms to evaporate from a surface.

LAMMPS also allows use of triclinic (non-orthogonal) simulation boxes. See the *Howto triclinic* page for a description of both general and restricted triclinic boxes and how to define them. General triclinic boxes (arbitrary edge vectors **A**, **B**, and **C**) are converted internally to restricted triclinic boxes with tilt factors (xy,xz,yz) which skew an otherwise orthogonal box.

The boundary <boundary> command settings explained above for the 6 faces of an orthogonal box also apply in similar manner to the 6 faces of a restricted triclinic box (and thus to the corresponding 6 faces of a general triclinic box), with the following context.

if the second dimension of a tilt factor (e.g. y for xy) is periodic, then the periodicity is enforced with the tilt factor offset. This means that for y periodicity a particle which exits the lower y boundary is displaced in the x-direction by xy before it re-enters the upper y boundary. And vice versa if a particle exits the upper y boundary. Likewise the ghost

atoms surrounding a particle near the lower y boundary include images of particles near the upper y-boundary which are displaced in the x-direction by xy. Similar rules apply for z-periodicity and the xz and/or yz tilt factors.

If the first dimension of a tilt factor is shrink-wrapped, then the shrink wrapping is applied to the tilted box face, to encompass the atoms. E.g. for a positive xy tilt, the xlo and xhi faces of the box are planes tilting in the +y direction as y increases. The position of these tilted planes are adjusted dynamically to shrink-wrap around the atoms to determine the xlo and xhi extents of the box.

1.10.4 Restrictions

This command cannot be used after the simulation box is defined by a *read_data* or *create_box* command or *read_restart* command. See the *change_box* command for how to change the simulation box boundaries after it has been defined.

For 2d simulations, the z dimension must be periodic.

1.10.5 Related commands

See the *thermo_modify* command for a discussion of lost atoms.

1.10.6 Default

```
boundary p p p
```

1.11 change_box command

1.11.1 Syntax

```
change_box group-ID parameter args ... keyword args ...
```

- group-ID = ID of group of atoms to (optionally) displace
- one or more parameter/arg pairs may be appended

parameter = x or y or z or xy or xz or yz or boundary or ortho or triclinic or set or remap
  x, y, z args = style value(s)
    style = final or delta or scale or volume
      final values = lo hi
        lo hi = box boundaries after displacement (distance units)
      delta values = dlo dhi
        dlo dhi = change in box boundaries after displacement (distance units)
      scale values = factor
        factor = multiplicative factor for change in box length after displacement
      volume value = none = adjust this dim to preserve volume of system
  xy, xz, yz args = style value
    style = final or delta
      final value = tilt
        tilt = tilt factor after displacement (distance units)
      delta value = dtilt
        dtilt = change in tilt factor after displacement (distance units)

boundary args = x y z
x,y,z = p or s or f or m, one or two letters
p is periodic
f is non-periodic and fixed
s is non-periodic and shrink-wrapped
m is non-periodic and shrink-wrapped with a minimum value
ortho args = none = change box to orthogonal
triclinic args = none = change box to triclinic
set args = none = store state of current box
remap args = none = remap atom coords from last saved state to current box

- zero or more keyword/value pairs may be appended
- keyword = *units*

units value = lattice or box
lattice = distances are defined in lattice units
box = distances are defined in simulation box units

1.11.2 Examples

```
change_box all xy final -2.0 z final 0.0 5.0 boundary p p f remap units box
change_box all x scale 1.1 y volume z volume remap
```

1.11.3 Description

Change the volume and/or shape and/or boundary conditions for the simulation box. Orthogonal simulation boxes have 3 adjustable size parameters (x,y,z). Triclinic (non-orthogonal) simulation boxes have 6 adjustable size/shape parameters (x,y,z,xy,xz,yz). Any or all of them can be adjusted independently by this command. Thus it can be used to expand or contract a box, or to apply a shear strain to a non-orthogonal box. It can also be used to change the boundary conditions for the simulation box, similar to the *boundary* command.

The size and shape of the initial simulation box are specified by the *create_box* or *read_data* or *read_restart* command used to setup the simulation. The size and shape may be altered by subsequent runs, e.g. by use of the *fix npt* or *fix deform* commands. The *create_box*, *read data*, and *read_restart* commands also determine whether the simulation box is orthogonal or triclinic and their doc pages explain the meaning of the xy,xz,yz tilt factors.

See the *Howto triclinic* page for a geometric description of triclinic boxes, as defined by LAMMPS, and how to transform these parameters to and from other commonly used triclinic representations.

The keywords used in this command are applied sequentially to the simulation box and the atoms in it, in the order specified.

Before the sequence of keywords are invoked, the current box size/shape is stored, in case a *remap* keyword is used to map the atom coordinates from a previously stored box size/shape to the current one.

After all the keywords have been processed, any shrink-wrap boundary conditions are invoked (see the *boundary* command) which may change simulation box boundaries, and atoms are migrated to new owning processors.

Note

This means that you cannot use the change_box command to enlarge a shrink-wrapped box, e.g. to make room to insert more atoms via the *create_atoms* command, because the simulation box will be re-shrink-wrapped before

the change_box command completes. Instead you could do something like this, assuming the simulation box is non-periodic and atoms extend from 0 to 20 in all dimensions:

```
change_box all x final -10 20
create_atoms 1 single -5 5 5       # this will fail to insert an atom

change_box all x final -10 20 boundary f s s
create_atoms 1 single -5 5 5
change_box all boundary s s s      # this will work
```

Note

Unlike the earlier "displace_box" version of this command, atom remapping is NOT performed by default. This command allows remapping to be done in a more general way, exactly when you specify it (zero or more times) in the sequence of transformations. Thus if you do not use the *remap* keyword, atom coordinates will not be changed even if the box size/shape changes. If a uniformly strained state is desired, the *remap* keyword should be specified.

Note

It is possible to lose atoms with this command. E.g. by changing the box without remapping the atoms, and having atoms end up outside of non-periodic boundaries. It is also possible to alter bonds between atoms straddling a boundary in bad ways. E.g. by converting a boundary from periodic to non-periodic. It is also possible when remapping atoms to put them (nearly) on top of each other. E.g. by converting a boundary from non-periodic to periodic. All of these will typically lead to bad dynamics and/or generate error messages.

Note

The simulation box size/shape can be changed by arbitrarily large amounts by this command. This is not a problem, except that the mapping of processors to the simulation box is not changed from its initial 3d configuration; see the *processors* command. Thus, if the box size/shape changes dramatically, the mapping of processors to the simulation box may not end up as optimal as the initial mapping attempted to be. You may wish to re-balance the atoms by using the *balance* command if that is the case.

Note

You cannot use this command after reading a restart file (and before a run is performed) if the restart file stored per-atom information from a fix and any of the specified keywords change the box size or shape or boundary conditions. This is because atoms may be moved to new processors and the restart info will not migrate with them. LAMMPS will generate an error if this could happen. Only the *ortho* and *triclinic* keywords do not trigger this error. One solution is to perform a "run 0" command before using the change_box command. This clears the per-atom restart data, whether it has been re-assigned to a new fix or not.

Note

Because the keywords used in this command are applied one at a time to the simulation box and the atoms in it, care

must be taken with triclinic cells to avoid exceeding the limits on skew after each transformation in the sequence. If skew is exceeded before the final transformation this can be avoided by changing the order of the sequence, or breaking the transformation into two or more smaller transformations. For more information on the allowed limits for box skew see the discussion on triclinic boxes on *Howto triclinic* doc page.

For the *x*, *y*, and *z* parameters, this is the meaning of their styles and values.

For style *final*, the final lo and hi box boundaries of a dimension are specified. The values can be in lattice or box distance units. See the discussion of the units keyword below.

For style *delta*, plus or minus changes in the lo/hi box boundaries of a dimension are specified. The values can be in lattice or box distance units. See the discussion of the units keyword below.

For style *scale*, a multiplicative factor to apply to the box length of a dimension is specified. For example, if the initial box length is 10, and the factor is 1.1, then the final box length will be 11. A factor less than 1.0 means compression.

The *volume* style changes the specified dimension in such a way that the overall box volume remains constant with respect to the operation performed by the preceding keyword. The *volume* style can only be used following a keyword that changed the volume, which is any of the *x*, *y*, *z* keywords. If the preceding keyword "key" had a *volume* style, then both it and the current keyword apply to the keyword preceding "key". I.e. this sequence of keywords is allowed:

```
change_box all x scale 1.1 y volume z volume
```

The *volume* style changes the associated dimension so that the overall box volume is unchanged relative to its value before the preceding keyword was invoked.

If the following command is used, then the z box length will shrink by the same 1.1 factor the x box length was increased by:

```
change_box all x scale 1.1 z volume
```

If the following command is used, then the y,z box lengths will each shrink by sqrt(1.1) to keep the volume constant. In this case, the y,z box lengths shrink so as to keep their relative aspect ratio constant:

```
change_box all x scale 1.1 y volume z volume
```

If the following command is used, then the final box will be a factor of 10% larger in x and y, and a factor of 21% smaller in z, so as to keep the volume constant:

```
change_box all x scale 1.1 z volume y scale 1.1 z volume
```

Note

For solids or liquids, when one dimension of the box is expanded, it may be physically undesirable to hold the other 2 box lengths constant since that implies a density change. For solids, adjusting the other dimensions via the *volume* style may make physical sense (just as for a liquid), but may not be correct for materials and potentials whose Poisson ratio is not 0.5.

For the *scale* and *volume* styles, the box length is expanded or compressed around its mid point.

For the *xy*, *xz*, and *yz* parameters, this is the meaning of their styles and values. Note that changing the tilt factors of a triclinic box does not change its volume.

For style *final*, the final tilt factor is specified. The value can be in lattice or box distance units. See the discussion of the units keyword below.

For style *delta*, a plus or minus change in the tilt factor is specified. The value can be in lattice or box distance units. See the discussion of the units keyword below.

All of these styles change the xy, xz, yz tilt factors. In LAMMPS, tilt factors (xy,xz,yz) for triclinic boxes are required to be no more than half the distance of the parallel box length. For example, if xlo = 2 and xhi = 12, then the x box length is 10 and the xy tilt factor must be between -5 and 5. Similarly, both xz and yz must be between -(xhi-xlo)/2 and +(yhi-ylo)/2. Note that this is not a limitation, since if the maximum tilt factor is 5 (as in this example), then configurations with tilt = ..., -15, -5, 5, 15, 25, ... are all equivalent. Any tilt factor specified by this command must be within these limits.

The *boundary* keyword takes arguments that have exactly the same meaning as they do for the *boundary* command. In each dimension, a single letter assigns the same style to both the lower and upper face of the box. Two letters assigns the first style to the lower face and the second style to the upper face.

The style *p* means the box is periodic; the other styles mean non-periodic. For style *f*, the position of the face is fixed. For style *s*, the position of the face is set so as to encompass the atoms in that dimension (shrink-wrapping), no matter how far they move. For style *m*, shrink-wrapping occurs, but is bounded by the current box edge in that dimension, so that the box will become no smaller. See the *boundary* command for more explanation of these style options.

Note that the "boundary" command itself can only be used before the simulation box is defined via a *read_data* or *create_box* or *read_restart* command. This command allows the boundary conditions to be changed later in your input script. Also note that the *read_restart* will change boundary conditions to match what is stored in the restart file. So if you wish to change them, you should use the change_box command after the read_restart command.

Note

Changing a periodic boundary to a non-periodic one will also cause the image flag for that dimension of all atoms to be reset to 0. LAMMPS will print a warning message, if that happens. Please note that this reset can lead to undesired changes when atoms are involved in bonded interactions that straddle periodic boundaries and thus the values of the image flag differs for those atoms.

The *ortho* and *triclinic* keywords convert the simulation box to be orthogonal or triclinic (non-orthogonal).

The simulation box is defined as either orthogonal or triclinic when it is created via the *create_box*, *read_data*, or *read_restart* commands.

These keywords allow you to toggle the existing simulation box from orthogonal to triclinic and vice versa. For example, an initial equilibration simulation can be run in an orthogonal box, the box can be toggled to triclinic, and then a *non-equilibrium MD (NEMD) simulation* can be run with deformation via the *fix deform* command.

If the simulation box is currently triclinic and has non-zero tilt in xy, yz, or xz, then it cannot be converted to an orthogonal box.

The *set* keyword saves the current box size/shape. This can be useful if you wish to use the *remap* keyword more than once or if you wish it to be applied to an intermediate box size/shape in a sequence of keyword operations. Note that the box size/shape is saved before any of the keywords are processed, i.e. the box size/shape at the time the create_box command is encountered in the input script.

The *remap* keyword remaps atom coordinates from the last saved box size/shape to the current box state. For example, if you stretch the box in the x dimension or tilt it in the xy plane via the *x* and *xy* keywords, then the *remap* command will dilate or tilt the atoms to conform to the new box size/shape, as if the atoms moved with the box as it deformed.

Note that this operation is performed without regard to periodic boundaries. Also, any shrink-wrapping of non-periodic boundaries (see the *boundary* command) occurs after all keywords, including this one, have been processed.

Only atoms in the specified group are remapped.

The *units* keyword determines the meaning of the distance units used to define various arguments. A *box* value selects standard distance units as defined by the *units* command, e.g. Angstroms for units = real or metal. A *lattice* value means the distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacing.

1.11.4 Restrictions

If you use the *ortho* or *triclinic* keywords, then at the point in the input script when this command is issued, no *dumps* can be active, nor can a *fix deform* be active. This is because these commands test whether the simulation box is orthogonal when they are first issued. Note that these commands can be used in your script before a change_box command is issued, so long as an *undump* or *unfix* command is also used to turn them off.

1.11.5 Related commands

fix deform, *boundary*

1.11.6 Default

The option default is units = lattice.

1.12 clear command

1.12.1 Syntax

```
clear
```

1.12.2 Examples

```
# (commands for 1st simulation)
clear
# (commands for 2nd simulation)
```

1.12.3 Description

This command deletes all atoms, restores all settings to their default values, and frees all memory allocated by LAMMPS. Once a clear command has been executed, it is almost as if LAMMPS were starting over, with only the exceptions noted below. This command enables multiple jobs to be run sequentially from one input script.

These settings are not affected by a clear command: the working directory (*shell* command), log file status (*log* command), echo status (*echo* command), and input script variables except for *atomfile* style variables (*variable* command).

1.12.4 Restrictions

none

1.12.5 Related commands

none

1.12.6 Default

none

1.13 comm_modify command

1.13.1 Syntax

```
comm_modify keyword value ...
```

- one or more keyword/value pairs may be appended
- keyword = *mode* or *cutoff* or *cutoff/multi* or *group* or *reduce/multi* or *vel*

  mode value = single, multi, or multi/old = communicate atoms within a single or multiple distances
  cutoff value = Rcut (distance units) = communicate atoms from this far away
  cutoff/multi collection value
    collection = atom collection or collection range (supports asterisk notation)
    value = Rcut (distance units) = communicate atoms for selected types from this far away
  reduce/multi arg = none = reduce number of communicated ghost atoms for multi style
  cutoff/multi/old type value
    type = atom type or type range (supports asterisk notation)
    value = Rcut (distance units) = communicate atoms for selected types from this far away
  group value = group-ID = only communicate atoms in the group
  vel value = yes or no = do or do not communicate velocity info with ghost atoms

1.13.2 Examples

```
comm_modify mode multi reduce/multi
comm_modify mode multi group solvent
comm_modify mode multi cutoff/multi 1 10.0 cutoff/multi 2*4 15.0
comm_modify vel yes
comm_modify mode single cutoff 5.0 vel yes
comm_modify cutoff/multi * 0.0
```

1.13.3 Description

This command sets parameters that affect the inter-processor communication of atom information that occurs each timestep as coordinates and other properties are exchanged between neighboring processors and stored as properties of ghost atoms.

Note

These options apply to the currently defined comm style. When you specify a *comm_style* or *read_restart* command, all communication settings are restored to their default or stored values, including those previously reset by a comm_modify command. Thus if your input script specifies a comm_style or read_restart command, you should use the comm_modify command after it.

The *mode* keyword determines whether a single or multiple cutoff distances are used to determine which atoms to communicate.

The default mode is *single* which means each processor acquires information for ghost atoms that are within a single distance from its subdomain. The distance is by default the maximum of the neighbor cutoff across all atom type pairs.

For many systems this is an efficient algorithm, but for systems with widely varying cutoffs for different type pairs, the *multi* or *multi/old* mode can be faster. In *multi*, each atom is assigned to a collection which should correspond to a set of atoms with similar interaction cutoffs. See the *neighbor* command for a detailed description of collections. In this case, each atom collection is assigned its own distance cutoff for communication purposes, and fewer atoms will be communicated. in *multi/old*, a similar technique is used but atoms are grouped by atom type. See the *neighbor multi* and *neighbor multi/old* commands for neighbor list construction options that may also be beneficial for simulations of this kind. The *multi* communication mode is only compatible with the *multi* neighbor style. The *multi/old* communication mode is comparable with both the *multi* and *multi/old* neighbor styles.

The *cutoff* keyword allows you to extend the ghost cutoff distance for communication mode *single*, which is the distance from the borders of a processor's subdomain at which ghost atoms are acquired from other processors. By default the ghost cutoff = neighbor cutoff = pairwise force cutoff + neighbor skin. See the *neighbor* command for more information about the skin distance. If the specified Rcut is greater than the neighbor cutoff, then extra ghost atoms will be acquired. If the provided cutoff is smaller, the provided value will be ignored, the ghost cutoff is set to the neighbor cutoff and a warning will be printed. Specifying a cutoff value of 0.0 will reset any previous value to the default. If bonded interactions exist and equilibrium bond length information is available, then also a heuristic based on that bond length is computed. It is used as communication cutoff, if there is no pair style present and no *comm_modify cutoff* command used. Otherwise a warning is printed, if this bond based estimate is larger than the communication cutoff used.

The *cutoff/multi* option is equivalent to *cutoff*, but applies to communication mode *multi* instead. Since the communication cutoffs are determined per atom collections, a collection specifier is needed and cutoff for one or multiple collections can be extended. Also ranges of collections using the usual asterisk notation can be given. Collections are indexed from 1 to N where N is the total number of collections. Note that the arguments for *cutoff/multi* are parsed right before each simulation to account for potential changes in the number of collections. Custom cutoffs are preserved between runs but if collections are redefined, one may want to re-specify the communication cutoffs. For granular pair

styles,the default cutoff is set to the sum of the current maximum atomic radii for each collection. The *cutoff/multi/old* option is similar to *cutoff/multi* except it operates on atom types as opposed to collections.

The *reduce/multi* option applies to *multi* and sets the communication cutoff for a particle equal to the maximum interaction distance between particles in the same collection. This reduces the number of ghost atoms that need to be communicated. This method is only compatible with the *multi* neighbor style and requires a half neighbor list and Newton on. See the *neighbor multi* command for more information.

These are simulation scenarios in which it may be useful or even necessary to set a ghost cutoff > neighbor cutoff:

- a single polymer chain with bond interactions, but no pairwise interactions
- bonded interactions (e.g. dihedrals) extend further than the pairwise cutoff
- ghost atoms beyond the pairwise cutoff are needed for some computation

In the first scenario, a pairwise potential is not defined. Thus the pairwise neighbor cutoff will be 0.0. But ghost atoms are still needed for computing bond, angle, etc interactions between atoms on different processors, or when the interaction straddles a periodic boundary.

The appropriate ghost cutoff depends on the *newton bond* setting. For newton bond *off*, the distance needs to be the furthest distance between any two atoms in the bond, angle, etc. E.g. the distance between 1-4 atoms in a dihedral. For newton bond *on*, the distance between the central atom in the bond, angle, etc and any other atom is sufficient. E.g. the distance between 2-4 atoms in a dihedral.

In the second scenario, a pairwise potential is defined, but its neighbor cutoff is not sufficiently long enough to enable bond, angle, etc terms to be computed. As in the previous scenario, an appropriate ghost cutoff should be set.

In the last scenario, a *fix* or *compute* or *pairwise potential* needs to calculate with ghost atoms beyond the normal pairwise cutoff for some computation it performs (e.g. locate neighbors of ghost atoms in a manybody pair potential). Setting the ghost cutoff appropriately can ensure it will find the needed atoms.

Note

In these scenarios, if you do not set the ghost cutoff long enough, and if there is only one processor in a periodic dimension (e.g. you are running in serial), then LAMMPS may "find" the atom it is looking for (e.g. the partner atom in a bond), that is on the far side of the simulation box, across a periodic boundary. This will typically lead to bad dynamics (i.e. the bond length is now the simulation box length). To detect if this is happening, see the *neigh_modify cluster* command.

The *group* keyword will limit communication to atoms in the specified group. This can be useful for models where no ghost atoms are needed for some kinds of particles. All atoms (not just those in the specified group) will still migrate to new processors as they move. The group specified with this option must also be specified via the *atom_modify first* command.

The *vel* keyword enables velocity information to be communicated with ghost particles. Depending on the *atom_style*, velocity info includes the translational velocity, angular velocity, and angular momentum of a particle. If the *vel* option is set to *yes*, then ghost atoms store these quantities; if *no* then they do not. The *yes* setting is needed by some pair styles which require the velocity state of both the I and J particles to compute a pairwise I,J interaction, as well as by some compute and fix commands.

Note that if the *fix deform* command is being used with its "remap v" option enabled, then the velocities for ghost atoms (in the fix deform group) mirrored across a periodic boundary will also include components due to any velocity shift that occurs across that boundary (e.g. due to dilation or shear).

1.13.4 Restrictions

Communication mode *multi* is currently only available for *comm_style* *brick*.

1.13.5 Related commands

comm_style, *neighbor*

1.13.6 Default

The option defaults are mode = single, group = all, cutoff = 0.0, vel = no. The cutoff default of 0.0 means that ghost cutoff = neighbor cutoff = pairwise force cutoff + neighbor skin.

1.14 comm_style command

1.14.1 Syntax

```
comm_style style
```

- style = *brick* or *tiled*

1.14.2 Examples

```
comm_style brick
comm_style tiled
```

1.14.3 Description

This command sets the style of inter-processor communication of atom information that occurs each timestep as coordinates and other properties are exchanged between neighboring processors and stored as properties of ghost atoms.

For the default *brick* style, the domain decomposition used by LAMMPS to partition the simulation box must be a regular 3d grid of bricks, one per processor. Each processor communicates with its 6 Cartesian neighbors in the grid to acquire information for nearby atoms.

For the *tiled* style, a more general domain decomposition can be used, as triggered by the *balance* or *fix balance* commands. The simulation box can be partitioned into non-overlapping rectangular-shaped "tiles" or varying sizes and shapes. Again there is one tile per processor. To acquire information for nearby atoms, communication must now be done with a more complex pattern of neighboring processors.

Note that this command does not actually define a partitioning of the simulation box (a domain decomposition), rather it determines what kinds of decompositions are allowed and the pattern of communication used to enable the decomposition. A decomposition is created when the simulation box is first created, via the *create_box* or *read_data* or *read_restart* commands. For both the *brick* and *tiled* styles, the initial decomposition will be the same, as described by *create_box* and *processors* commands. The decomposition can be changed via the *balance* or *fix balance* commands.

1.14.4 Restrictions

none

1.14.5 Related commands

comm_modify, *processors*, *balance*, *fix balance*

1.14.6 Default

The default style is brick.

1.15 compute command

1.15.1 Syntax

```
compute ID group-ID style args
```

- ID = user-assigned name for the computation
- group-ID = ID of the group of atoms to perform the computation on
- style = one of a list of possible style names (see below)
- args = arguments used by a particular style

1.15.2 Examples

```
compute 1 all temp
compute newtemp flow temp/partial 1 1 0
compute 3 all ke/atom
```

1.15.3 Description

Define a diagnostic computation that will be performed on a group of atoms. Quantities calculated by a compute are instantaneous values, meaning they are calculated from information about atoms on the current timestep or iteration, though internally a compute may store some information about a previous state of the system. Defining a compute does not perform the computation. Instead computes are invoked by other LAMMPS commands as needed (e.g., to calculate a temperature needed for a thermostat fix or to generate thermodynamic or dump file output). See the *Howto output* page for a summary of various LAMMPS output options, many of which involve computes.

The ID of a compute can only contain alphanumeric characters and underscores.

Computes calculate and store any of four *styles* of quantities: global, per-atom, local, or per-grid.

A global quantity is one or more system-wide values, e.g. the temperature of the system. A per-atom quantity is one or more values per atom, e.g. the kinetic energy of each atom. Per-atom values are set to 0.0 for atoms not in the specified compute group. Local quantities are calculated by each processor based on the atoms it owns, but there may be zero or more per atom, e.g. a list of bond distances. Per-grid quantities are calculated on a regular 2d or 3d grid which overlays

a 2d or 3d simulation domain. The grid points and the data they store are distributed across processors; each processor owns the grid points which fall within its subdomain.

As a general rule of thumb, computes that produce per-atom quantities have the word "atom" at the end of their style, e.g. *ke/atom*. Computes that produce local quantities have the word "local" at the end of their style, e.g. *bond/local*. Computes that produce per-grid quantities have the word "grid" at the end of their style, e.g. *property/grid*. And styles with neither "atom" or "local" or "grid" at the end of their style name produce global quantities.

Global, per-atom, local, and per-grid quantities can also be of three *kinds*: a single scalar value (global only), a vector of values, or a 2d array of values. For per-atom, local, and per-grid quantities, a "vector" means a single value for each atom, each local entity (e.g. bond), or grid cell. Likewise an "array", means multiple values for each atom, each local entity, or each grid cell.

Note that a single compute can produce any combination of global, per-atom, local, or per-grid values. Likewise it can produce any combination of scalar, vector, or array output for each style. The exception is that for per-atom, local, and per-grid output, either a vector or array can be produced, but not both. The doc page for each compute explains the values it produces.

When a compute output is accessed by another input script command it is referenced via the following bracket notation, where ID is the ID of the compute:

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

In other words, using one bracket reduces the dimension of the quantity once (vector → scalar, array → vector). Using two brackets reduces the dimension twice (array → scalar). Thus, for example, a command that uses global scalar compute values as input can also process elements of a vector or array. Depending on the command, this can either be done directly using the syntax in the table, or by first defining a *variable* of the appropriate style to store the quantity, then using the variable as an input to the command.

Note that commands and *variables* which take compute outputs as input typically do not allow for all styles and kinds of data (e.g., a command may require global but not per-atom values, or it may require a vector of values, not a scalar). This means there is typically no ambiguity about referring to a compute output as c_ID even if it produces, for example, both a scalar and vector. The doc pages for various commands explain the details, including how any ambiguities are resolved.

In LAMMPS, the values generated by a compute can be used in several ways:

- The results of computes that calculate a global temperature or pressure can be used by fixes that do thermostatting or barostatting or when atom velocities are created.
- Global values can be output via the *thermo_style custom* or *fix ave/time* command. Or the values can be referenced in a *variable equal* or *variable atom* command.
- Per-atom values can be output via the *dump custom* command. Or they can be time-averaged via the *fix ave/atom* command or reduced by the *compute reduce* command. Or the per-atom values can be referenced in an *atom-style variable*.
- Local values can be reduced by the *compute reduce* command, or histogrammed by the *fix ave/histo* command, or output by the *dump local* command.

The results of computes that calculate global quantities can be either "intensive" or "extensive" values. Intensive means the value is independent of the number of atoms in the simulation (e.g., temperature). Extensive means the value scales with the number of atoms in the simulation (e.g., total rotational kinetic energy). *Thermodynamic output* will normalize extensive values by the number of atoms in the system, depending on the "thermo_modify norm" setting. It will not

normalize intensive values. If a compute value is accessed in another way (e.g., by a *variable*), you may want to know whether it is an intensive or extensive value. See the page for individual computes for further info.

LAMMPS creates its own computes internally for thermodynamic output. Three computes are always created, named "thermo_temp", "thermo_press", and "thermo_pe", as if these commands had been invoked in the input script:

```
compute thermo_temp all temp
compute thermo_press all pressure thermo_temp
compute thermo_pe all pe
```

Additional computes for other quantities are created if the thermo style requires it. See the documentation for the *thermo_style* command.

Fixes that calculate temperature or pressure, i.e. for thermostatting or barostatting, may also create computes. These are discussed in the documentation for specific *fix* commands.

In all these cases, the default computes LAMMPS creates can be replaced by computes defined by the user in the input script, as described by the *thermo_modify* and *fix modify* commands.

Properties of either a default or user-defined compute can be modified via the *compute_modify* command.

Computes can be deleted with the *uncompute* command.

Code for new computes can be added to LAMMPS; see the *Modify* page for details. The results of their calculations accessed in the various ways described above.

Each compute style has its own page which describes its arguments and what it does. Here is an alphabetic list of compute styles available in LAMMPS. They are also listed in more compact form on the *Commands compute* doc page.

There are also additional accelerated compute styles included in the LAMMPS distribution for faster performance on CPUs, GPUs, and KNLs. The individual style names on the *Commands compute* page are followed by one or more of (g,i,k,o,t) to indicate which accelerated styles exist.

- *ackland/atom* - determines the local lattice structure based on the Ackland formulation
- *adf* - angular distribution function of triples of atoms
- *aggregate/atom* - aggregate ID for each atom
- *angle* - energy of each angle sub-style
- *angle/local* - theta and energy of each angle
- *angmom/chunk* - angular momentum for each chunk
- *ave/sphere/atom* - compute local density and temperature around each atom
- *basal/atom* - calculates the hexagonal close-packed "c" lattice vector of each atom
- *body/local* - attributes of body sub-particles
- *bond* - energy of each bond sub-style
- *bond/local* - distance and energy of each bond
- *born/matrix* - second derivative or potential with respect to strain
- *centro/atom* - centro-symmetry parameter for each atom
- *centroid/stress/atom* - centroid based stress tensor for each atom

- *chunk/atom* - assign chunk IDs to each atom
- *chunk/spread/atom* - spreads chunk values to each atom in chunk
- *cluster/atom* - cluster ID for each atom
- *cna/atom* - common neighbor analysis (CNA) for each atom
- *cnp/atom* - common neighborhood parameter (CNP) for each atom
- *com* - center of mass of group of atoms
- *com/chunk* - center of mass for each chunk
- *contact/atom* - contact count for each spherical particle
- *coord/atom* - coordination number for each atom
- *count/type* - count of atoms or bonds by type
- *damage/atom* - Peridynamic damage for each atom
- *dihedral* - energy of each dihedral sub-style
- *dihedral/local* - angle of each dihedral
- *dilatation/atom* - Peridynamic dilatation for each atom
- *dipole* - dipole vector and total dipole
- *dipole/chunk* - dipole vector and total dipole for each chunk
- *dipole/tip4p* - dipole vector and total dipole with TIP4P pair style
- *dipole/tip4p/chunk* - dipole vector and total dipole for each chunk with TIP4P pair style
- *displace/atom* - displacement of each atom
- *dpd* - total values of internal conductive energy, internal mechanical energy, chemical energy, and harmonic average of internal temperature
- *dpd/atom* - per-particle values of internal conductive energy, internal mechanical energy, chemical energy, and internal temperature
- *edpd/temp/atom* - per-atom temperature for each eDPD particle in a group
- *efield/atom* - electric field at each atom
- *efield/wolf/atom* - electric field at each atom
- *entropy/atom* - pair entropy fingerprint of each atom
- *erotate/asphere* - rotational energy of aspherical particles
- *erotate/rigid* - rotational energy of rigid bodies
- *erotate/sphere* - rotational energy of spherical particles
- *erotate/sphere/atom* - rotational energy for each spherical particle
- *event/displace* - detect event on atom displacement
- *fabric* - calculates fabric tensors from pair interactions
- *fep* - compute free energies for alchemical transformation from perturbation theory
- *fep/ta* - compute free energies for a test area perturbation
- *force/tally* - force between two groups of atoms via the tally callback mechanism
- *fragment/atom* - fragment ID for each atom

- *global/atom* - assign global values to each atom from arrays of global values
- *group/group* - energy/force between two groups of atoms
- *gyration* - radius of gyration of group of atoms
- *gyration/chunk* - radius of gyration for each chunk
- *gyration/shape* - shape parameters from gyration tensor
- *gyration/shape/chunk* - shape parameters from gyration tensor for each chunk
- *heat/flux* - heat flux through a group of atoms
- *heat/flux/tally* - heat flux through a group of atoms via the tally callback mechanism
- *heat/flux/virial/tally* - virial heat flux between two groups via the tally callback mechanism
- *hexorder/atom* - bond orientational order parameter q6
- *hma* - harmonically mapped averaging for atomic crystals
- *improper* - energy of each improper sub-style
- *improper/local* - angle of each improper
- *inertia/chunk* - inertia tensor for each chunk
- *ke* - translational kinetic energy
- *ke/atom* - kinetic energy for each atom
- *ke/atom/eff* - per-atom translational and radial kinetic energy in the electron force field model
- *ke/eff* - kinetic energy of a group of nuclei and electrons in the electron force field model
- *ke/rigid* - translational kinetic energy of rigid bodies
- *composition/atom* - local composition for each atom
- *mliap* - gradients of energy and forces with respect to model parameters and related quantities for training machine learning interatomic potentials
- *momentum* - translational momentum
- *msd* - mean-squared displacement of group of atoms
- *msd/chunk* - mean-squared displacement for each chunk
- *msd/nongauss* - MSD and non-Gaussian parameter of group of atoms
- *nbond/atom* - calculates number of bonds per atom
- *omega/chunk* - angular velocity for each chunk
- *orientorder/atom* - Steinhardt bond orientational order parameters Ql
- *pace* - atomic cluster expansion descriptors and related quantities
- *pair* - values computed by a pair style
- *pair/local* - distance/energy/force of each pairwise interaction
- *pe* - potential energy
- *pe/atom* - potential energy for each atom
- *pe/mol/tally* - potential energy between two groups of atoms separated into intermolecular and intramolecular components via the tally callback mechanism
- *pe/tally* - potential energy between two groups of atoms via the tally callback mechanism

- *plasticity/atom* - Peridynamic plasticity for each atom
- *pod/atom* - POD descriptors for each atom
- *podd/atom* - derivative of POD descriptors for each atom
- *pod/local* - local POD descriptors and their derivatives
- *pod/global* - global POD descriptors and their derivatives
- *pressure* - total pressure and pressure tensor
- *pressure/alchemy* - mixed system total pressure and pressure tensor for *fix alchemy* runs
- *pressure/uef* - pressure tensor in the reference frame of an applied flow field
- *property/atom* - convert atom attributes to per-atom vectors/arrays
- *property/chunk* - extract various per-chunk attributes
- *property/grid* - convert per-grid attributes to per-grid vectors/arrays
- *property/local* - convert local attributes to local vectors/arrays
- *ptm/atom* - determines the local lattice structure based on the Polyhedral Template Matching method
- *rattlers/atom* - identify under-coordinated rattler atoms
- *rdf* - radial distribution function $g(r)$ histogram of group of atoms
- *reaxff/atom* - extract ReaxFF bond information
- *reduce* - combine per-atom quantities into a single global value
- *reduce/chunk* - reduce per-atom quantities within each chunk
- *reduce/region* - same as compute reduce, within a region
- *rheo/property/atom* - convert atom attributes in RHEO package to per-atom vectors/arrays
- *rigid/local* - extract rigid body attributes
- *saed* - electron diffraction intensity on a mesh of reciprocal lattice nodes
- *slcsa/atom* - perform Supervised Learning Crystal Structure Analysis (SL-CSA)
- *slice* - extract values from global vector or array
- *smd/contact/radius* - contact radius for Smooth Mach Dynamics
- *smd/damage* - damage status of SPH particles in Smooth Mach Dynamics
- *smd/hourglass/error* - error associated with approximated relative separation in Smooth Mach Dynamics
- *smd/internal/energy* - per-particle enthalpy in Smooth Mach Dynamics
- *smd/plastic/strain* - equivalent plastic strain per particle in Smooth Mach Dynamics
- *smd/plastic/strain/rate* - time rate of the equivalent plastic strain in Smooth Mach Dynamics
- *smd/rho* - per-particle mass density in Smooth Mach Dynamics
- *smd/tlsph/defgrad* - deformation gradient in Smooth Mach Dynamics
- *smd/tlsph/dt* - CFL-stable time increment per particle in Smooth Mach Dynamics
- *smd/tlsph/num/neighs* - number of particles inside the smoothing kernel radius for Smooth Mach Dynamics
- *smd/tlsph/shape* - current shape of the volume of a particle for Smooth Mach Dynamics
- *smd/tlsph/strain* - Green–Lagrange strain tensor for Smooth Mach Dynamics

- *smd/tlsph/strain/rate* - rate of strain for Smooth Mach Dynamics
- *smd/tlsph/stress* - per-particle Cauchy stress tensor for SPH particles
- *smd/triangle/vertices* - coordinates of vertices corresponding to the triangle elements of a mesh for Smooth Mach Dynamics
- *smd/ulsph/effm* - per-particle effective shear modulus
- *smd/ulsph/num/neighs* - number of neighbor particles inside the smoothing kernel radius for Smooth Mach Dynamics
- *smd/ulsph/strain* - logarithmic strain tensor for Smooth Mach Dynamics
- *smd/ulsph/strain/rate* - logarithmic strain rate for Smooth Mach Dynamics
- *smd/ulsph/stress* - per-particle Cauchy stress tensor and von Mises equivalent stress in Smooth Mach Dynamics
- *smd/vol* - per-particle volumes and their sum in Smooth Mach Dynamics
- *snap* - gradients of SNAP energy and forces with respect to linear coefficients and related quantities for fitting SNAP potentials
- *sna/atom* - bispectrum components for each atom
- *sna/grid* - global array of bispectrum components on a regular grid
- *sna/grid/local* - local array of bispectrum components on a regular grid
- *snad/atom* - derivative of bispectrum components for each atom
- *snav/atom* - virial contribution from bispectrum components for each atom
- *sph/e/atom* - per-atom internal energy of Smooth-Particle Hydrodynamics atoms
- *sph/rho/atom* - per-atom density of Smooth-Particle Hydrodynamics atoms
- *sph/t/atom* - per-atom internal temperature of Smooth-Particle Hydrodynamics atoms
- *spin* - magnetic quantities for a system of atoms having spins
- *stress/atom* - stress tensor for each atom
- *stress/cartesian* - stress tensor in cartesian coordinates
- *stress/cylinder* - stress tensor in cylindrical coordinates
- *stress/mop* - normal components of the local stress tensor using the method of planes
- *stress/mop/profile* - profile of the normal components of the local stress tensor using the method of planes
- *stress/spherical* - stress tensor in spherical coordinates
- *stress/tally* - stress between two groups of atoms via the tally callback mechanism
- *tdpd/cc/atom* - per-atom chemical concentration of a specified species for each tDPD particle
- *temp* - temperature of group of atoms
- *temp/asphere* - temperature of aspherical particles
- *temp/body* - temperature of body particles
- *temp/chunk* - temperature of each chunk
- *temp/com* - temperature after subtracting center-of-mass velocity
- *temp/cs* - temperature based on the center-of-mass velocity of atom pairs that are bonded to each other
- *temp/deform* - temperature excluding box deformation velocity

- *temp/deform/eff* - temperature excluding box deformation velocity in the electron force field model
- *temp/drude* - temperature of Core–Drude pairs
- *temp/eff* - temperature of a group of nuclei and electrons in the electron force field model
- *temp/partial* - temperature excluding one or more dimensions of velocity
- *temp/profile* - temperature excluding a binned velocity profile
- *temp/ramp* - temperature excluding ramped velocity component
- *temp/region* - temperature of a region of atoms
- *temp/region/eff* - temperature of a region of nuclei and electrons in the electron force field model
- *temp/rotate* - temperature of a group of atoms after subtracting out their center-of-mass and angular velocities
- *temp/sphere* - temperature of spherical particles
- *temp/uef* - kinetic energy tensor in the reference frame of an applied flow field
- *ti* - thermodynamic integration free energy values
- *torque/chunk* - torque applied on each chunk
- *vacf* - velocity auto-correlation function of group of atoms
- *vcm/chunk* - velocity of center-of-mass for each chunk
- *viscosity/cos* - velocity profile under cosine-shaped acceleration
- *voronoi/atom* - Voronoi volume and neighbors for each atom
- *xrd* - X-ray diffraction intensity on a mesh of reciprocal lattice nodes

1.15.4 Restrictions

none

1.15.5 Related commands

uncompute, *compute_modify*, *fix ave/atom*, *fix ave/time*, *fix ave/histo*

1.15.6 Default

none

1.16 compute_modify command

1.16.1 Syntax

```
compute_modify compute-ID keyword value ...
```

- compute-ID = ID of the compute to modify
- one or more keyword/value pairs may be listed
- keyword = *extra/dof* or *dynamic/dof*

extra/dof value = N
  N = # of extra degrees of freedom to subtract
dynamic/dof value = yes or no
  yes/no = do or do not re-compute the number of degrees of freedom (DOF) contributing to the temperature

1.16.2 Examples

```
compute_modify myTemp extra/dof 0
compute_modify newtemp dynamic/dof yes extra/dof 600
```

1.16.3 Description

Modify one or more parameters of a previously defined compute. Not all compute styles support all parameters.

The *extra/dof* keyword refers to how many degrees of freedom are subtracted (typically from 3*N*) as a normalizing factor in a temperature computation. Only computes that compute a temperature use this option. The default is 2 or 3 for *2d or 3d systems* which is a correction factor for an ensemble of velocities with zero total linear momentum. For compute temp/partial, if one or more velocity components are excluded, the value used for *extra/dof* is scaled accordingly. You can use a negative number for the *extra/dof* parameter if you need to add degrees-of-freedom. See the *compute temp/asphere* command for an example.

The *dynamic/dof* keyword determines whether the number of atoms *N* in the compute group and their associated degrees of freedom (DOF) are re-computed each time a temperature is computed. Only compute styles that calculate a temperature use this option. By default, *N* and their DOF are assumed to be constant. If you are adding atoms or molecules to the system (see the *fix pour*, *fix deposit*, and *fix gcmc* commands) or expect atoms or molecules to be lost (e.g. due to exiting the simulation box or via *fix evaporate*), then this option should be used to ensure the temperature is correctly normalized.

1.16.4 Restrictions

none

1.16.5 Related commands

compute

1.16.6 Default

The option defaults are extra/dof = 2 or 3 for 2d or 3d systems, respectively, and dynamic/dof = *no*.

1.17 create_atoms command

1.17.1 Syntax

```
create_atoms type style args keyword values ...
```

- type = atom type (1-Ntypes or type label) of atoms to create (offset for molecule creation)
- style = *box* or *region* or *single* or *mesh* or *random*

  box args = none
  region args = region-ID
    region-ID = particles will only be created if contained in the region
  single args = x y z
    x,y,z = coordinates of a single particle (distance units)
  mesh args = STL-file
    STL-file = file with triangle mesh in STL format
  random args = N seed region-ID
    N = number of particles to create
    seed = random # seed (positive integer)
    region-ID = create atoms within this region, use NULL for entire simulation box

- zero or more keyword/value pairs may be appended
- keyword = *mol* or *basis* or *ratio* or *subset* or *remap* or *var* or *set* or *radscale* or *meshmode* or *rotate* or *overlap* or *maxtry* or *units*

  mol values = template-ID seed
    template-ID = ID of molecule template specified in a separate molecule command
    seed = random # seed (positive integer)
  basis values = M itype
    M = which basis atom
    itype = atom type (1-Ntypes or type label) to assign to this basis atom
  ratio values = frac seed
    frac = fraction of lattice sites (0 to 1) to populate randomly
    seed = random # seed (positive integer)
  subset values = Nsubset seed
    Nsubset = # of lattice sites to populate randomly
    seed = random # seed (positive integer)
  remap value = yes or no
  var value = name = variable name to evaluate for test of atom creation
  set values = dim name
    dim = x or y or z
    name = name of variable to set with x, y, or z atom position
  radscale value = factor
    factor = scale factor for setting atom radius
  meshmode values = mode arg
    mode = bisect or qrand
    bisect arg = radthresh
      radthresh = threshold value for mesh to determine when to split triangles (distance units)
    qrand arg = density
      density = minimum number density for atoms place on mesh triangles (inverse distance squared units)
  rotate values = theta Rx Ry Rz
    theta = rotation angle for single molecule (degrees)

Rx,Ry,Rz = rotation vector for single molecule
overlap value = Doverlap
Doverlap = only insert if at least this distance from all existing atoms
maxtry value = Ntry
Ntry = number of attempts to insert a particle before failure
units value = lattice or box
lattice = the geometry is defined in lattice units
box = the geometry is defined in simulation box units

1.17.2 Examples

```
create_atoms 1 box

labelmap atom 1 Pt
create_atoms Pt box

labelmap atom 1 C 2 Si
create_atoms C region regsphere basis Si C

create_atoms 3 region regsphere basis 2 3
create_atoms 3 region regsphere basis 2 3 ratio 0.5 74637
create_atoms 3 single 0 0 5
create_atoms 1 box var v set x xpos set y ypos
create_atoms 2 random 50 12345 NULL overlap 2.0 maxtry 50
create_atoms 1 mesh open_box.stl meshmode qrand 0.1 units box
create_atoms 1 mesh funnel.stl meshmode bisect 4.0 units box radscale 0.9
```

1.17.3 Description

This command creates atoms (or molecules) within the simulation box, either on a lattice, or at random points, or on a surface defined by a triangulated mesh. Or it creates a single atom (or molecule) at a specified point. It is an alternative to reading in atom coordinates explicitly via a *read_data* or *read_restart* command.

To use this command a simulation box must already exist, which is typically created via the *create_box* command. Before using this command, a lattice must typically also be defined using the *lattice* command, unless you specify the *single* or *mesh* style with units = box or the *random* style. To create atoms on a lattice for general triclinic boxes, see the discussion below.

For the remainder of this doc page, a created atom or molecule is referred to as a "particle".

If created particles are individual atoms, they are assigned the specified atom *type*, though this can be altered via the *basis* keyword as discussed below. If molecules are being created, the type of each atom in the created molecule is specified in a specified file read by the *molecule* command, and those values are added to the specified atom *type* (e.g., if *type* = 2 and the file specifies atom types 1, 2, and 3, then each created molecule will have atom types 3, 4, and 5).

Note

You cannot use this command to create atoms that are outside the simulation box; they will just be ignored by LAMMPS. This is true even if you are using shrink-wrapped box boundaries, as specified by the *boundary* command. However, you can first use the *change_box* command to temporarily expand the box, then add atoms via create_atoms, then finally use change_box command again if needed to re-shrink-wrap the new atoms. See the

change_box doc page for an example of how to do this, using the create_atoms *single* style to insert a new atom outside the current simulation box.

For the *box* style, the create_atoms command fills the entire simulation box with particles on the lattice. If your simulation box is periodic, you should ensure its size is a multiple of the lattice spacings, to avoid unwanted atom overlaps at the box boundaries. If your box is periodic and a multiple of the lattice spacing in a particular dimension, LAMMPS is careful to put exactly one particle at the boundary (on either side of the box), not zero or two.

For the *region* style, a geometric volume is filled with particles on the lattice. This volume is what is both inside the simulation box and also consistent with the region volume. See the *region* command for details. Note that a region can be specified so that its "volume" is either inside or outside its geometric boundary. Also note that if a region is the same size as a periodic simulation box (in some dimension), LAMMPS does NOT implement the same logic described above for the *box* style, to ensure exactly one particle at periodic boundaries. If this is desired, you should either use the *box* style, or tweak the region size to get precisely the particles you want.

If the simulation box is formulated as a general triclinic box defined by arbitrary edge vectors **A**, **B**, **C**, then the *box* and *region* styles will create atoms on a lattice commensurate with those edge vectors. See the *Howto_triclinic* doc page for a detailed explanation of orthogonal, restricted triclinic, and general triclinic simulation boxes. As with the *create_box* command, the *lattice* command used by this command must be of style *custom* and use its *triclinic/general* option. The *a1, *a2*, *a3* settings of the *lattice* command define the edge vectors of a unit cell of the general triclinic lattice. The *create_box* command creates a simulation box which replicates that unit cell along each of the **A**, **B**, **C** edge vectors.

Note

LAMMPS allows specification of general triclinic simulation boxes as a convenience for users who may be converting data from solid-state crystallographic representations or from DFT codes for input to LAMMPS. However, as explained on the *Howto_triclinic* doc page, internally, LAMMPS only uses restricted triclinic simulation boxes. This means the box created by the *create_box* command as well as the atoms created by this command with their per-atom information (e.g. coordinates, velocities) are converted (rotated) from general to restricted triclinic form when the two commands are invoked. The *Howto_triclinic* doc page also discusses other LAMMPS commands which can input/output general triclinic representations of the simulation box and per-atom data.

The *box* style will fill the entire general triclinic box with particles on the lattice, as explained above.

Note

The *region* style also operates as explained above, but the check for particles inside the region is performed *after* the particle coordinates have been converted to the restricted triclinic box. This means the region must also be defined with respect to the restricted triclinic box, not the general triclinic box.

If the simulation box is general triclinic, the *single*, *random*, and *mesh* styles described next operate on the box *after* it has been converted to restricted triclinic. So all the settings for those styles should be made in that context.

For the *single* style, a single particle is added to the system at the specified coordinates. This can be useful for debugging purposes or to create a tiny system with a handful of particles at specified positions. For a 2d simulation the specified z coordinate must be 0.0.

Changed in version 2Jun2022.

The *porosity* style has been renamed to *random* with added functionality.

For the *random* style, *N* particles are added to the system at randomly generated coordinates, which can be useful for generating an amorphous system. For 2d simulations, the z coordinates of all added atoms will be 0.0.

The particles are created one by one using the specified random number *seed*, resulting in the same set of particle coordinates, independent of how many processors are being used in the simulation. Unless the *overlap* keyword is specified, particles created by the *random* style will typically be highly overlapped. Various additional criteria can be used to accept or reject a random particle insertion; see the keyword discussion below. Multiple attempts per particle are made (see the *maxtry* keyword) until the insertion is either successful or fails. If this command fails to add all requested *N* particles, a warning will be output.

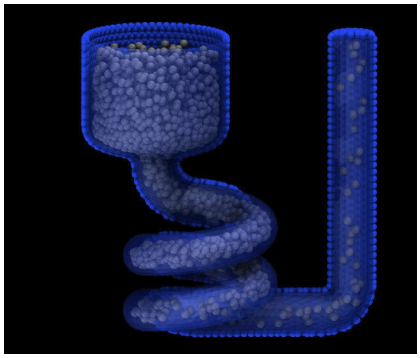
If the *region-ID* argument is specified as NULL, then the randomly created particles will be anywhere in the simulation box. If a *region-ID* is specified, a geometric volume is filled that is both inside the simulation box and is also consistent with the region volume. See the *region* command for details. Note that a region can be specified so that its "volume" is either inside or outside its geometric boundary.

Note that the create_atoms command adds particles to those that already exist. This means it can be used to add particles to a system previously read in from a data or restart file. Or the create_atoms command can be used multiple times, to add multiple sets of particles to the simulation. For example, grain boundaries can be created, by interleaving the create_atoms command with *lattice* commands specifying different orientations.

When this command is used, care should be taken to ensure the resulting system does not contain particles that are highly overlapped. Such overlaps will cause many interatomic potentials to compute huge energies and forces, leading to bad dynamics. There are several strategies to avoid this problem:

- Use the *delete_atoms overlap* command after create_atoms. For example, this strategy can be used to overlay and surround a large protein molecule with a volume of water molecules, then delete water molecules that overlap with the protein atoms.
- For the *random* style, use the optional *overlap* keyword to avoid overlaps when each new particle is created.
- Before running dynamics on an overlapped system, perform an *energy minimization*. Or run initial dynamics with *pair_style soft* or with *fix nve/limit* to un-overlap the particles, before running normal dynamics.

Added in version 2Jun2022.



For the *mesh* style, a file with a triangle mesh in STL format is read and one or more particles are placed into the area of each triangle. The reader supports both ASCII and binary files conforming to the format on the Wikipedia page. Binary STL files (e.g. as frequently offered for 3d-printing) can also be first converted to ASCII for editing with the *stl_bin2txt tool*. The use of the *units box* option is required. There are two algorithms available for placing atoms: *bisect* and *qrand*. They can be selected via the *meshmode* option; *bisect* is the default. If the atom style allows it, the radius will be set to a value depending on the algorithm and the value of the *radscale* parameter (see below), and the atoms created from the mesh are assigned a new molecule ID.

In *bisect* mode a particle is created at the center of each triangle unless the average distance of the triangle vertices from its center is larger than the *radthresh* value (default is lattice spacing in x-direction). In case the average distance is over the threshold, the triangle is recursively split into two halves along the the longest side until the threshold is reached. There will be at least one sphere per triangle. The value of *radthresh* is set as an argument to *meshmode bisect*. The average distance of the vertices from the center is also used to set the radius.

In *qrand* mode a quasi-random sequence is used to distribute particles on mesh triangles using an approach by *(Roberts)*. Particles are added to the triangle until the minimum number density is met or exceeded such that every triangle will have at least one particle. The minimum number density is set as an argument to the *qrand* option. The radius will be

set so that the sum of the area of the radius of the particles created in place of a triangle will be equal to the area of that triangle.

Note

The atom placement algorithms in the *mesh* style benefit from meshes where triangles are close to equilateral. It is therefore recommended to pre-process STL files to optimize the mesh accordingly. There are multiple open source and commercial software tools available with the capability to generate optimized meshes.

Note

In most cases the atoms created in *mesh* style will become an immobile or rigid object that would not be time integrated or moved by *fix move* or *fix rigid*. For computational efficiency *and* to avoid undesired contributions to pressure and potential energy due to close contacts, it is usually beneficial to exclude computing interactions between the created particles using *neigh_modify exclude*.

Individual atoms are inserted by this command, unless the *mol* keyword is used. It specifies a *template-ID* previously defined using the *molecule* command, which reads a file that defines the molecule. The coordinates, atom types, charges, etc, as well as any bond/angle/etc and special neighbor information for the molecule can be specified in the molecule file. See the *molecule* command for details. The only settings required to be in this file are the coordinates and types of atoms in the molecule.

Note

If you are using the *mol* keyword in combination with the *atom style template* command, they must use the same molecule template-ID.

Using a lattice to add molecules, e.g. via the *box* or *region* or *single* styles, is exactly the same as adding atoms on lattice points, except that entire molecules are added at each point, i.e. on the point defined by each basis atom in the unit cell as it tiles the simulation box or region. This is done by placing the geometric center of the molecule at the lattice point, and (by default) giving the molecule a random orientation about the point. The random *seed* specified with the *mol* keyword is used for this operation, and the random numbers generated by each processor are different. This means the coordinates of individual atoms (in the molecules) will be different when running on different numbers of processors, unlike when atoms are being created in parallel.

Note that with random rotations, it may be important to use a lattice with a large enough spacing that adjacent molecules will not overlap, regardless of their relative orientations. See the description of the *rotate* keyword below, which overrides the default random orientation and inserts all molecules at a specified orientation.

Note

If the *create_box* command is used to create the simulation box, followed by the create_atoms command with its *mol* option for adding molecules, then you typically need to use the optional keywords allowed by the *create_box* command for extra bonds (angles,etc) or extra special neighbors. This is because by default, the *create_box* command sets up a non-molecular system that does not allow molecules to be added.

This is the meaning of the other optional keywords.

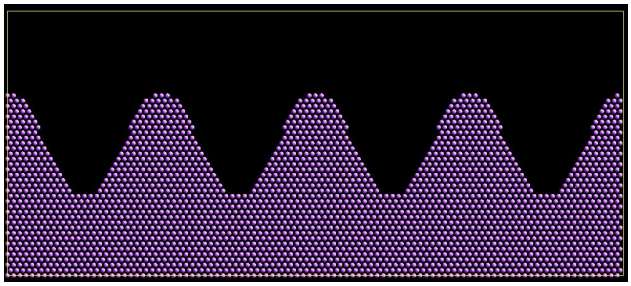
The *basis* keyword is only used when atoms (not molecules) are being created. It specifies an atom type that will be assigned to specific basis atoms as they are created. See the *lattice* command for specifics on how basis atoms are defined for the unit cell of the lattice. By default, all created atoms are assigned the argument *type* as their atom type.

The *ratio* and *subset* keywords can be used in conjunction with the *box* or *region* styles to limit the total number of particles inserted. The lattice defines a set of *Nlatt* eligible sites for inserting particles, which may be limited by the *region* style or the *var* and *set* keywords. For the *ratio* keyword, only the specified fraction of them ($0 \leq f \leq 1$) will be assigned particles. For the *subset* keyword only the specified *Nsubset* of them will be assigned particles. In both cases the assigned lattice sites are chosen randomly. An iterative algorithm is used that ensures the correct number of particles are inserted, in a perfectly random fashion. Which lattice sites are selected will change with the number of processors used.

The *remap* keyword only applies to the *single* style. If it is set to *yes*, then if the specified position is outside the simulation box, it will mapped back into the box, assuming the relevant dimensions are periodic. If it is set to *no*, no remapping is done and no particle is created if its position is outside the box.

The *var* and *set* keywords can be used together to provide a criterion for accepting or rejecting the addition of an individual atom, based on its coordinates. They apply to all styles except *single*. The *name* specified for the *var* keyword is the name of an *equal-style variable* that should evaluate to a zero or non-zero value based on one or two or three variables that will store the x, y, or z coordinates of an atom (one variable per coordinate). If used, these other variables must be *internal-style variables* defined in the input script; their initial numeric value can be anything. They must be internal-style variables, because this command resets their values directly. The *set* keyword is used to identify the names of these other variables, one variable for the x-coordinate of a created atom, one for y, and one for z.

When an atom is created, its (x, y, z) coordinates become the values for any *set* variable that is defined. The *var* variable is then evaluated. If the returned value is 0.0, the atom is not created. If it is non-zero, the atom is created.



As an example, these commands can be used in a 2d simulation, to create a sinusoidal surface. Note that the surface is "rough" due to individual lattice points being "above" or "below" the mathematical expression for the sinusoidal curve. If a finer lattice were used, the sinusoid would appear to be "smoother". Also note the use of the "xlat" and "ylat" *thermo_style* keywords, which converts lattice spacings to distance.

```
dimension   2
variable    x equal 100
variable    y equal 25
lattice     hex 0.8442
region      box block 0 $x 0 $y -0.5 0.5
create_box  1 box

variable    xx internal 0.0
variable    yy internal 0.0
variable    v equal "(0.2*v_y*ylat * cos(v_xx/xlat * 2.0*PI*4.0/v_x) + 0.5*v_y*ylat - v_yy) > 0.0"
create_atoms  1 box var v set x xx set y yy
write_dump  all atom sinusoid.lammpstrj
```

The *rotate* keyword allows specification of the orientation at which molecules are inserted. The axis of rotation is determined by the rotation vector (R_x, R_y, R_z) that goes through the insertion point. The specified *theta* determines the angle of rotation around that axis. Note that the direction of rotation for the atoms around the rotation axis is consistent with the right-hand rule: if your right-hand's thumb points along R, then your fingers wrap around the axis in the direction of rotation.

The *radscale* keyword only applies to the *mesh* style and adjusts the radius of created particles (see above), provided this is supported by the atom style. Its value is a prefactor (must be > 0.0, default is 1.0) that is applied to the atom radius inferred from the size of the individual triangles in the triangle mesh that the particle corresponds to.

Added in version 2Jun2022.

The *overlap* keyword only applies to the *random* style. It prevents newly created particles from being created closer than the specified *Doverlap* distance from any other particle. If particles have finite size (see *atom_style sphere* for example) *Doverlap* should be specified large enough to include the particle size in the non-overlapping criterion. If molecules are being randomly inserted, then an insertion is only accepted if each particle in the molecule meets the overlap criterion with respect to other particles (not including particles in the molecule itself).

Note

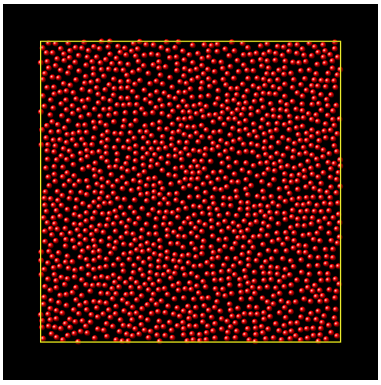
Checking for overlaps is a costly $\mathcal{O}(N(N+M))$ operation for inserting N new particles into a system with M existing particles. This is because distances to all M existing particles are computed for each new particle that is added. Thus the intended use of this keyword is to add relatively small numbers of particles to systems that remain at a relatively low density even after the new particles are created. Careful use of the *maxtry* keyword in combination with *overlap* is recommended. See the discussion above about systems with overlapped particles for alternate strategies that allow for overlapped insertions.

Added in version 2Jun2022.

The *maxtry* keyword only applies to the *random* style. It limits the number of attempts to generate valid coordinates for a single new particle that satisfy all requirements imposed by the *region*, *var*, and *overlap* keywords. The default is 10 attempts per particle before the loop over the requested N particles advances to the next particle. Note that if insertion success is unlikely (e.g., inserting new particles into a dense system using the *overlap* keyword), setting the *maxtry* keyword to a large value may result in this command running for a long time.

Here is an example for the *random* style using these commands

```
units          lj
dimension      2
region         box block 0 50 0 50 -0.5 0.5
create_box     1 box
create_atoms   1 random 2000 13487 NULL overlap 1.0 maxtry 50
pair_style     lj/cut 2.5
pair_coeff     1 1 1.0 1.0 2.5
```



to produce a system as shown in the image with 1520 particles (out of 2000 requested) that are moderately dense and which have no overlaps sufficient to prevent the LJ pair_style from running properly (because the overlap criterion is 1.0). The create_atoms command ran for 0.3 s on a single CPU core.

The *units* keyword determines the meaning of the distance units used by parameters for various styles. A *box* value selects standard distance units as defined by the *units* command (e.g., Å for units = *real* or *metal*. A *lattice* value means the distance units are in lattice spacings. These are affected settings:

- for *single* style: coordinates of the particle created
- for *random* style: overlap distance *Doverlap* by the *overlap* keyword
- for *mesh* style: *bisect* threshold value for *meshmode* = *bisect*
- for *mesh* style: *radthresh* value for *meshmode* = *bisect*
- for *mesh* style: *density* value for *meshmode* = *qrand*

Since *density* represents an area (distance ^2), the lattice spacing factor is also squared.

Atom IDs are assigned to created atoms in the following way. The collection of created atoms are assigned consecutive IDs that start immediately following the largest atom ID existing before the create_atoms command was invoked. This is done by the processor's communicating the number of atoms they each own, the first processor numbering its atoms from 1 to N_1, the second processor from $N_1 + 1$ to N_2, and so on, where N_1 is the number of atoms owned by the first processor, N_2 is the number owned by the second processor, and so forth. Thus, when the same simulation is performed on different numbers of processors, there is no guarantee a particular created atom will be assigned the same ID in both simulations. If molecules are being created, molecule IDs are assigned to created molecules in a similar fashion.

Aside from their ID, atom type, and *xyz* position, other properties of created atoms are set to default values, depending on which quantities are defined by the chosen *atom style*. See the *atom style* command for more details. See the *set* and *velocity* commands for info on how to change these values.

- charge = 0.0
- dipole moment magnitude = 0.0
- diameter = 1.0
- shape = 0.0 0.0 0.0
- density = 1.0
- volume = 1.0
- velocity = 0.0 0.0 0.0
- angular velocity = 0.0 0.0 0.0
- angular momentum = 0.0 0.0 0.0
- quaternion = (1,0,0,0)
- bonds, angles, dihedrals, impropers = none

If molecules are being created, these defaults can be overridden by values specified in the file read by the *molecule* command. That is, the file typically defines bonds (angles, etc.) between atoms in the molecule, and can optionally define charges on each atom.

Note that the *sphere* atom style sets the default particle diameter to 1.0 as well as the density. This means the mass for the particle is not 1.0, but is $\frac{\pi}{6}d^3 = 0.5236$, where d is the diameter. When using the *mesh* style, the particle diameter is adjusted from the size of the individual triangles in the triangle mesh.

Note that the *ellipsoid* atom style sets the default particle shape to (0.0 0.0 0.0) and the density to 1.0, which means it is a point particle, not an ellipsoid, and has a mass of 1.0.

Note that the *peri* style sets the default volume and density to 1.0 and thus also set the mass for the particle to 1.0.

The *set* command can be used to override many of these default settings.

1.17.4 Restrictions

An *atom_style* must be previously defined to use this command.

A rotation vector specified for a single molecule must be in the z-direction for a 2d model.

For *molecule templates* that are created from multiple files, i.e. contain multiple molecule *sets*, only the first set is used. To create multiple molecules the files currently need to be merged and different molecule IDs assigned with a Molecules section.

1.17.5 Related commands

lattice, *region*, *create_box*, *read_data*, *read_restart*

1.17.6 Default

The default for the *basis* keyword is that all created atoms are assigned the argument *type* as their atom type (when single atoms are being created). The other defaults are *remap* = no, *rotate* = random, *radscale* = 1.0, *radthresh* = x-lattice spacing, *overlap* not checked, *maxtry* = 10, and *units* = lattice.

(Roberts) R. Roberts (2019) "Evenly Distributing Points in a Triangle." Extreme Learning. http://extremelearning.com.au/evenly-distributing-points-in-a-triangle/

1.18 create_bonds command

1.18.1 Syntax

```
create_bonds style args ... keyword value ...
```

- style = *many* or *single/bond* or *single/angle* or *single/dihedral* or *single/improper*

```
many args = group-ID group2-ID btype rmin rmax
  group-ID = ID of first group
  group2-ID = ID of second group, bonds will be between atoms in the 2 groups
  btype = bond type of created bonds
  rmin = minimum distance between pair of atoms to bond together
  rmax = maximum distance between pair of atoms to bond together
single/bond args = btype batom1 batom2
  btype = bond type of new bond
  batom1,batom2 = atom IDs for two atoms in bond
single/angle args = atype aatom1 aatom2 aatom3
  atype = angle type of new angle
  aatom1,aatom2,aatom3 = atom IDs for three atoms in angle
single/dihedral args = dtype datom1 datom2 datom3 datom4
  dtype = dihedral type of new dihedral
  datom1,datom2,datom3,datom4 = atom IDs for four atoms in dihedral
single/improper args = itype iatom1 iatom2 iatom3 iatom4
  itype = improper type of new improper
  iatom1,iatom2,iatom3,iatom4 = atom IDs for four atoms in improper
```

- zero or more keyword/value pairs may be appended

- keyword = *special*

special value = yes or no

1.18.2 Examples

```
create_bonds many all all 1 1.0 1.2
create_bonds many surf solvent 3 2.0 2.4
create_bonds single/bond 1 1 2
create_bonds single/angle 5 52 98 107 special no
create_bonds single/dihedral 2 4 19 27 101
create_bonds single/improper 3 23 26 31 57
```

1.18.3 Description

Create bonds between pairs of atoms that meet a specified distance criteria. Or create a single bond, angle, dihedral or improper between 2, 3, or 4 specified atoms.

The new bond (angle, dihedral, improper) interactions will then be computed during a simulation by the bond (angle, dihedral, improper) potential defined by the *bond_style*, *bond_coeff*, *angle_style*, *angle_coeff*, *dihedral_style*, *dihedral_coeff*, *improper_style*, *improper_coeff* commands.

The *many* style is useful for adding bonds to a system (e.g., between nearest neighbors in a lattice of atoms) without having to enumerate all the bonds in the data file read by the *read_data* command.

The *single* styles are useful for adding bonds, angles, dihedrals, and impropers to a system incrementally, then continuing a simulation.

Note that this command does not auto-create any angle, dihedral, or improper interactions when a bond is added, nor does it auto-create any bonds when an angle, dihedral, or improper is added. It also will not auto-create any angles when a dihedral or improper is added. Thus, the flexibility of this command is limited. It can be used several times to create different types of bond at different distances, but it cannot typically auto-create all the bonds or angles or dihedrals or impropers that would normally be defined in a data file for a complex system of molecules.

Note

If the system has no bonds (angles, dihedrals, impropers) to begin with, or if more bonds per atom are being added than currently exist, then you must ensure that the number of bond types and the maximum number of bonds per atom are set to large enough values, and similarly for angles, dihedrals, impropers, and special neighbors, otherwise an error may occur when too many bonds (angles, dihedrals, impropers) are added to an atom. If the *read_data* command is used to define the system, these parameters can be set via its optional *extra/bond/types*, *extra/bond/per/atom*, and similar keywords to the command. If the *create_box* command is used to define the system, these two parameters can be set via its optional *bond/types* and *extra/bond/per/atom* arguments, and similarly for angles, dihedrals, and impropers. See the corresponding documentation pages for these two commands for details.

The *many* style will create bonds between pairs of atoms I, J, where I is in one of the two specified groups and J is in the other. The two groups can be the same (e.g., group "all"). The created bonds will be of bond type *btype*, where *btype* must be a value between 1 and the number of bond types defined.

For a bond to be created, an I, J pair of atoms must be a distance D apart such that $r_{\min} \leq D \leq r_{\max}$.

The following settings must have been made in an input script before the *many* style is used:

- special_bonds weight for 1–2 interactions must be 0.0
- a *pair_style* must be defined
- no *kspace_style* defined
- minimum *pair_style* cutoff + *neighbor* skin $\geq r_{\max}$

These settings are required so that a neighbor list can be created to search for nearby atoms. Pairs of atoms that are already bonded cannot appear in the neighbor list, to avoid creation of duplicate bonds. The neighbor list for all atom type pairs must also extend to a distance that encompasses the *rmax* for new bonds to create. When using periodic boundary conditions, the box length in each periodic dimension must be larger than *rmax*, so that no bonds are created between the system and its own periodic image.

Note

If you want to create bonds between pairs of 1–3 or 1–4 atoms in the current bond topology, then you need to use *special_bonds lj 0 1 1* to ensure those pairs appear in the neighbor list. They will not appear with the default special_bonds settings, which are zero for 1–2, 1–3, and 1–4 atoms. 1–3 or 1–4 atoms are those which are two hops or three hops apart in the bond topology.

An additional requirement for this style is that your system must be ready to perform a simulation. This means, for example, that all *pair_style* coefficients be set via the *pair_coeff* command. A *bond_style* command and all bond coefficients must also be set, even if no bonds exist before this command is invoked. This is because the building of neighbor list requires initialization and setup of a simulation, similar to what a *run* command would require.

Note that you can change any of these settings after this command executes (e.g., if you wish to use long-range Coulombic interactions) via the *kspace_style* command for your subsequent simulation.

The *single/bond* style creates a single bond of type *btype* between two atoms with IDs *batom1* and *batom2*. *Btype* must be a value between 1 and the number of bond types defined.

The *single/angle* style creates a single angle of type *atype* between three atoms with IDs *aatom1*, *aatom2*, and *aatom3*. The ordering of the atoms is the same as in the *Angles* section of a data file read by the *read_data* command (i.e., the three atoms are ordered linearly within the angle; the central atom is *aatom2*). *Atype* must be a value between 1 and the number of angle types defined.

The *single/dihedral* style creates a single dihedral of type *dtype* between four atoms with IDs *datom1*, *datom2*, *datom3*, and *datom4*. The ordering of the atoms is the same as in the *Dihedrals* section of a data file read by the *read_data* command. I.e. the 4 atoms are ordered linearly within the dihedral. *dtype* must be a value between 1 and the number of dihedral types defined.

The *single/improper* style creates a single improper of type *itype* between four atoms with IDs *iatom1*, *iatom2*, *iatom3*, and *iatom4*. The ordering of the atoms is the same as in the *Impropers* section of a data file read by the *read_data* command. I.e. the 4 atoms are ordered linearly within the improper. *itype* must be a value between 1 and the number of improper types defined.

The keyword *special* controls whether an internal list of special bonds is created after one or more bonds, or a single angle, dihedral, or improper is added to the system.

The default value is *yes*. A value of *no* cannot be used with the *many* style.

This is an expensive operation since the bond topology for the system must be walked to find all 1–2, 1–3, and 1–4 interactions to store in an internal list, which is used when pairwise interactions are weighted; see the *special_bonds* command for details.

Thus if you are adding a few bonds or a large list of angles all at the same time, by using this command repeatedly, it is more efficient to only trigger the internal list to be created once, after the last bond (or angle, or dihedral, or improper) is added:

```
create_bonds single/bond 5 52 98 special no
create_bonds single/bond 5 73 74 special no
...
create_bonds single/bond 5 17 386 special no
create_bonds single/bond 4 112 183 special yes
```

Note that you **must** ensure the internal list is rebuilt after the last bond (angle, dihedral, improper) is added, *before* performing a simulation. Otherwise, pairwise interactions will not be properly excluded or weighted. LAMMPS does **not** check that you have done this correctly.

1.18.4 Restrictions

This command cannot be used with molecular systems defined using molecule template files via the *molecule* and *atom_style template* commands.

For style *many*, no *kspace style* must be defined. Also, the *rmax* value must be smaller than any periodic box length and the neighbor list cutoff (largest pair cutoff plus neighbor skin).

1.18.5 Related commands

create_atoms, *delete_bonds*

1.18.6 Default

The keyword default is special = yes.

1.19 create_box command

1.19.1 Syntax

```
create_box N region-ID keyword value ...
create_box N NULL alo ahi blo bhi clo chi keyword value ...
```

- N = # of atom types to use in this simulation
- region-ID = ID of region to use as simulation domain or NULL for general triclinic box
- alo,ahi,blo,bhi,clo,chi = multipliers on a1,a2,a3 vectors defined by *lattice* command (only when region-ID = NULL)
- zero or more keyword/value pairs may be appended
- keyword = *bond/types* or *angle/types* or *dihedral/types* or *improper/types* or *extra/bond/per/atom* or *extra/angle/per/atom* or *extra/dihedral/per/atom* or *extra/improper/per/atom* or *extra/special/per/atom*

bond/types value = # of bond types
angle/types value = # of angle types
dihedral/types value = # of dihedral types
improper/types value = # of improper types
extra/bond/per/atom value = # of bonds per atom
extra/angle/per/atom value = # of angles per atom
extra/dihedral/per/atom value = # of dihedrals per atom
extra/improper/per/atom value = # of impropers per atom
extra/special/per/atom value = # of special neighbors per atom

1.19.2 Examples

```
# orthogonal or restricted triclinic box using regionID = mybox
create_box 2 mybox
create_box 2 mybox bond/types 2 extra/bond/per/atom 1
```

```
# 2d general triclinic box using primitive cell for 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
```

```
# 3d general triclinic box using primitive cell for 3d fcc lattice
lattice custom 1.0 a2 0.0 0.5 0.5 a1 0.5 0.0 0.5 a3 0.5 0.5 0.0 basis 0.0 0.0 0.0 triclinic/general
create box 1 NULL -5 5 -10 10 0 20
```

1.19.3 Description

This command creates a simulation box. It also partitions the box into a regular 3d grid of smaller sub-boxes, one per processor (MPI task). The geometry of the partitioning is based on the size and shape of the simulation box, the number of processors being used and the settings of the *processors* command. The partitioning can later be changed by the *balance* or *fix balance* commands.

Simulation boxes in LAMMPS can be either orthogonal or triclinic in shape. Orthogonal boxes are a brick in 3d (rectangle in 2d) with 6 faces that are each perpendicular to one of the standard xyz coordinate axes. Triclinic boxes are a parallelepiped in 3d (parallelogram in 2d) with opposite pairs of faces parallel to each other. LAMMPS supports two forms of triclinic boxes, restricted and general, which differ in how the box is oriented with respect to the xyz coordinate axes. See the *Howto triclinic* for a detailed description of all 3 kinds of simulation boxes.

The argument *N* is the number of atom types that will be used in the simulation.

Orthogonal and restricted triclinic boxes are created by specifying a region ID previously defined by the *region* command. General triclinic boxes are discussed below.

If the region is not of style *prism*, then LAMMPS encloses the region (block, sphere, etc.) with an axis-aligned orthogonal bounding box which becomes the simulation domain. For a 2d simulation, the zlo and zhi values of the simulation box must straddle zero.

If the region is of style *prism*, LAMMPS creates a non-orthogonal simulation domain shaped as a parallelepiped with triclinic symmetry. As defined by the *region prism* command, the parallelepiped has an "origin" at (xlo,ylo,zlo) and three edge vectors starting from the origin given by $\vec{a} = (x_{\rm hi} - x_{\rm lo}, 0, 0)$; $\vec{b} = (xy, y_{\rm hi} - y_{\rm lo}, 0)$; and $\vec{c} = (xz, yz, z_{\rm hi} - z_{\rm lo})$. In LAMMPS lingo, this is a restricted triclinic box because the three edge vectors cannot be defined in arbitrary (general) directions. The parameters xy, xz, and yz can be 0.0 or positive or negative values and are called "tilt factors"

because they are the amount of displacement applied to faces of an originally orthogonal box to transform it into the parallelepiped. For a 2d simulation, the zlo and zhi values of the simulation box must straddle zero.

Typically a *prism* region used with the create_box command should have tilt factors (xy, xz, yz) that do not skew the box more than half the distance of the parallel box length. For example, if $x_{\rm lo} = 2$ and $x_{\rm hi} = 12$, then the x box length is 10 and the xy tilt factor must be between -5 and 5. Similarly, both xz and yz must be between $-(x_{\rm hi} - x_{\rm lo})/2$ and $+(y_{\rm hi} - y_{\rm lo})/2$. Note that this is not a limitation, since if the maximum tilt factor is 5 (as in this example), then configurations with tilt = $\ldots, -15, -5, 5, 15, 25, \ldots$ are all geometrically equivalent.

LAMMPS will issue a warning if the tilt factors of the created box do not meet this criterion. This is because simulations with large tilt factors may run inefficiently, since they require more ghost atoms and thus more communication. With very large tilt factors, LAMMPS may eventually produce incorrect trajectories and stop with errors due to lost atoms or similar issues.

See the *Howto triclinic* page for geometric descriptions of triclinic boxes and tilt factors, as well as how to transform the restricted triclinic parameters to and from other commonly used triclinic representations.

When a prism region is used, the simulation domain should normally be periodic in the dimension that the tilt is applied to, which is given by the second dimension of the tilt factor (e.g., y for xy tilt). This is so that pairs of atoms interacting across that boundary will have one of them shifted by the tilt factor. Periodicity is set by the *boundary* command. For example, if the xy tilt factor is non-zero, then the y dimension should be periodic. Similarly, the z dimension should be periodic if xz or yz is non-zero. LAMMPS does not require this periodicity, but you may lose atoms if this is not the case.

Note that if your simulation will tilt the box (e.g., via the *fix deform* command), the simulation box must be created as triclinic, even if the tilt factors are initially 0.0. You can also change an orthogonal box to a triclinic box or vice versa by using the *change box* command with its *ortho* and *triclinic* options.

Note

If the system is non-periodic (in a dimension), then you should not make the lo/hi box dimensions (as defined in your *region* command) radically smaller/larger than the extent of the atoms you eventually plan to create (e.g., via the *create_atoms* command). For example, if your atoms extend from 0 to 50, you should not specify the box bounds as -10000 and 10000. This is because as described above, LAMMPS uses the specified box size to lay out the 3d grid of processors. A huge (mostly empty) box will be sub-optimal for performance when using "fixed" boundary conditions (see the *boundary* command). When using "shrink-wrap" boundary conditions (see the *boundary* command), a huge (mostly empty) box may cause a parallel simulation to lose atoms the first time that LAMMPS shrink-wraps the box around the atoms.

As noted above, general triclinic boxes in LAMMPS allow the box to have arbitrary edge vectors **A**, **B**, **C**. The only restrictions are that the three vectors be distinct, non-zero, and not co-planar. They must also define a right-handed system such that (**A** x **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.

To create a general triclinic boxes, the region is specified as NULL and the next 6 parameters (alo,ahi,blo,bhi,clo,chi) define the three edge vectors **A**, **B**, **C** using additional information previously defined by the *lattice* command.

The lattice must be of style *custom* and use its *triclinic/general* option. This insures the lattice satisfies the restrictions listed above. The *a1, *a2, a3* settings of the *lattice* command define the edge vectors of a unit cell of the general triclinic lattice. This command uses them to define the three edge vectors and origin of the general triclinic box as:

- **A** = (ahi-alo) * *a1*
- **B** = (bhi-blo) * *a2*
- **C** = (chi-clo) * *a3*

- origin = (alo*a1 + blo*a2 + clo*a3)

For 2d general triclinic boxes, clo = -0.5 and chi = 0.5 is required.

Note

LAMMPS allows specification of general triclinic simulation boxes as a convenience for users who may be converting data from solid-state crystallographic representations or from DFT codes for input to LAMMPS. However, as explained on the *Howto_triclinic* doc page, internally, LAMMPS only uses restricted triclinic simulation boxes. This means the box defined by this command and per-atom information (e.g. coordinates, velocities) defined by the *create_atoms* command are converted (rotated) from general to restricted triclinic form when the two commands are invoked. The <Howto_triclinic>` doc page also discusses other LAMMPS commands which can input/output general triclinic representations of the simulation box and per-atom data.

The optional keywords can be used to create a system that allows for bond (angle, dihedral, improper) interactions, or for molecules with special 1–2, 1–3, or 1–4 neighbors to be added later. These optional keywords serve the same purpose as the analogous keywords that can be used in a data file which are recognized by the *read_data* command when it sets up a system.

Note that if these keywords are not used, then the create_box command creates an atomic (non-molecular) simulation that does not allow bonds between pairs of atoms to be defined, or a *bond potential* to be specified, or for molecules with special neighbors to be added to the system by commands such as *create_atoms mol*, *fix deposit* or *fix pour*.

As an example, see the examples/deposit/in.deposit.molecule script, which deposits molecules onto a substrate. Initially there are no molecules in the system, but they are added later by the *fix deposit* command. The create_box command in the script uses the bond/types and extra/bond/per/atom keywords to allow this. If the added molecule contained more than one special bond (allowed by default), an extra/special/per/atom keyword would also need to be specified.

1.19.4 Restrictions

An *atom_style* and *region* must have been previously defined to use this command.

1.19.5 Related commands

read_data, *create_atoms*, *region*

1.19.6 Default

none

1.20 delete_atoms command

1.20.1 Syntax

```
delete_atoms style args keyword value ...
```

- style = *group* or *region* or *overlap* or *random* or *variable*

  group args = group-ID
  region args = region-ID
  overlap args = cutoff group1-ID group2-ID
    cutoff = delete one atom from pairs of atoms within the cutoff (distance units)
    group1-ID = one atom in pair must be in this group
    group2-ID = other atom in pair must be in this group
  random args = ranstyle value eflag group-ID region-ID seed
    ranstyle = fraction or count
      for fraction:
        value = fraction (0.0 to 1.0) of eligible atoms to delete
        eflag = no for fast approximate deletion, yes for exact deletion
      for count:
        value = number of atoms to delete
        eflag = no for warning if count > eligible atoms, yes for error
    group-ID = group within which to perform deletions
    region-ID = region within which to perform deletions
                or NULL to only impose the group criterion
    seed = random number seed (positive integer)
  variable args = variable-name

- zero or more keyword/value pairs may be appended
- keyword = *compress* or *bond* or *mol*

  compress value = no or yes
  bond value = no or yes
  mol value = no or yes

1.20.2 Examples

```
delete_atoms group edge
delete_atoms region sphere compress no
delete_atoms overlap 0.3 all all
delete_atoms overlap 0.5 solvent colloid
delete_atoms random fraction 0.1 yes all cube 482793 bond yes
delete_atoms random fraction 0.3 no polymer NULL 482793 bond yes
delete_atoms random count 500 no ions NULL 482793
delete_atoms variable checkers
```

1.20.3 Description

Delete the specified atoms. This command can be used, for example, to carve out voids from a block of material or to delete created atoms that are too close to each other (e.g., at a grain boundary).

For style *group*, all atoms belonging to the group are deleted.

For style *region*, all atoms in the region volume are deleted. Additional atoms can be deleted if they are in a molecule for which one or more atoms were deleted within the region; see the *mol* keyword discussion below.

For style *overlap* pairs of atoms whose distance of separation is within the specified cutoff distance are searched for, and one of the two atoms is deleted. Only pairs where one of the two atoms is in the first group specified and the other atom is in the second group are considered. The atom that is in the first group is the one that is deleted.

Note that it is OK for the two group IDs to be the same (e.g., group *all*), or for some atoms to be members of both groups. In these cases, either atom in the pair may be deleted. Also note that if there are atoms which are members of both groups, the only guarantee is that at the end of the deletion operation, enough deletions will have occurred that no atom pairs within the cutoff will remain (subject to the group restriction). There is no guarantee that the minimum number of atoms will be deleted, or that the same atoms will be deleted when running on different numbers of processors.

For style *random* a subset of eligible atoms are deleted. Which atoms to delete are chosen randomly using the specified random number *seed*. Which atoms are deleted may vary when running on different numbers of processors.

For *ranstyle* = *fraction*, the specified fractional *value* (0.0 to 1.0) of eligible atoms are deleted. If *eflag* is set to *no*, then the number of deleted atoms will be approximate, but the operation will be fast. If *eflag* is set to *yes*, then the number deleted will match the requested fraction, but for large systems the selection of deleted atoms may take additional time to determine.

For *ranstyle* = *count*, the specified integer *value* is the number of eligible atoms are deleted. If *eflag* is set to *no*, then if the requested number is larger then the number of eligible atoms, a warning is issued and only the eligible atoms are deleted instead of the requested *value*. If *eflag* is set to *yes*, an error is triggered instead and LAMMPS will exit. For large systems the selection of atoms to delete may take additional time to determine, the same as for requesting an exact fraction with *pstyle* = *fraction*.

Which atoms are eligible for deletion for style *random* is determined by the specified *group-ID* and *region-ID*. To be eligible, an atom must be in both the specified group and region. If *group-ID* = all, there is effectively no group criterion. If *region-ID* is specified as NULL, no region criterion is imposed.

Added in version 4May2022.

For style *variable*, all atoms for which the atom-style variable with the given name evaluates to non-zero will be deleted. Additional atoms can be deleted if they are in a molecule for which one or more atoms were deleted within the region; see the *mol* keyword discussion below. This option allows complex selections of atoms not covered by the other options listed above.

Here is the meaning of the optional keywords.

If the *compress* keyword is set to *yes*, then after atoms are deleted, then atom IDs are re-assigned so that they run from 1 to the number of atoms in the system. Note that this is not done for molecular systems (see the *atom_style* command), regardless of the *compress* setting, since it would foul up the bond connectivity that has already been assigned. However, the *reset_atoms id* command can be used after this command to accomplish the same thing.

Note that the re-assignment of IDs is not really a compression, where gaps in atom IDs are removed by decrementing atom IDs that are larger. Instead the IDs for all atoms are erased, and new IDs are assigned so that the atoms owned by individual processors have consecutive IDs, as the *create_atoms* command explains.

A molecular system with fixed bonds, angles, dihedrals, or improper interactions, is one where the topology of the interactions is typically defined in the data file read by the *read_data* command, and where the interactions themselves are defined with the *bond_style*, *angle_style*, etc. commands. If you delete atoms from such a system, you must be

careful not to end up with bonded interactions that are stored by remaining atoms but which include deleted atoms. This will cause LAMMPS to generate a "missing atoms" error when the bonded interaction is computed. The *bond* and *mol* keywords offer two ways to do that.

It the *bond* keyword is set to *yes* then any bond or angle or dihedral or improper interaction that includes a deleted atom is also removed from the lists of such interactions stored by non-deleted atoms. Note that simply deleting interactions due to dangling bonds (e.g., at a surface) may result in a inaccurate or invalid model for the remaining atoms.

It the *mol* keyword is set to *yes*, then for every atom that is deleted, all other atoms in the same molecule (with the same molecule ID) will also be deleted. This is not done for atoms with molecule ID = 0, since such an ID is assumed to flag isolated atoms that are not part of molecules.

Note

The molecule deletion operation is invoked after all individual atoms have been deleted using the rules described above for each style. This means additional atoms may be deleted that are not in the group or region, that are not required by the overlap cutoff criterion, or that will create a higher fraction of porosity than was requested.

1.20.4 Restrictions

The *overlap* styles requires inter-processor communication to acquire ghost atoms and build a neighbor list. This means that your system must be ready to perform a simulation before using this command (force fields setup, atom masses set, etc.). Since a neighbor list is used to find overlapping atom pairs, it also means that you must define a *pair style* with the minimum force cutoff distance between any pair of atoms types (plus the *neighbor* skin) $\geq$ the specified overlap cutoff.

If the *special_bonds* command is used with a setting of 0, then a pair of bonded atoms (1–2, 1–3, or 1–4) will not appear in the neighbor list, and thus will not be considered for deletion by the *overlap* styles. You probably do not want to delete one atom in a bonded pair anyway.

The *bond yes* option cannot be used with molecular systems defined using molecule template files via the *molecule* and *atom_style template* commands.

1.20.5 Related commands

create_atoms, *reset_atoms id*

1.20.6 Default

The option defaults are compress = yes, bond = no, mol = no.

1.21 delete_bonds command

1.21.1 Syntax

```
delete_bonds group-ID style arg keyword ...
```

- group-ID = group ID
- style = *multi* or *atom* or *bond* or *angle* or *dihedral* or *improper* or *stats*

multi arg = none
atom arg = an atom type or range of types (see below)
bond arg = a bond type or range of types (see below)
angle arg = an angle type or range of types (see below)
dihedral arg = a dihedral type or range of types (see below)
improper arg = an improper type or range of types (see below)
stats arg = none

- zero or more keywords may be appended
- keyword = *any* or *undo* or *remove* or *special*

any arg = none = turn off interactions if any atoms are in the group (or on if undo is also used)
undo arg = none = turn specified bonds on instead of off
remove arg = permanently remove bonds that have been turned off
special arg = recompute pairwise 1-2, 1-3, and 1-4 lists

1.21.2 Examples

```
delete_bonds frozen multi remove
delete_bonds all atom 4 special
delete_bonds all bond 0*3 special
delete_bonds all stats

labelmap atom 4 hc
delete_bonds all atom hc special
```

1.21.3 Description

Turn off (or on) molecular topology interactions (i.e., bonds, angles, dihedrals, and/or impropers). This command is useful for deleting interactions that have been previously turned off by bond-breaking potentials. It is also useful for turning off topology interactions between frozen or rigid atoms. Pairwise interactions can be turned off via the *neigh_modify exclude* command. The *fix shake* command also effectively turns off certain bond and angle interactions.

For all styles, by default, an interaction is only turned off (or on) if all the atoms involved are in the specified group. See the *any* keyword to change the behavior.

Several of the styles (*atom*, *bond*, *angle*, *dihedral*, *improper*) take a *type* as an argument. The specified *type* can be a *type label*. Otherwise, the type should be an integer from 0 to *N*, where *N* is the number of relevant types (atom types, bond types, etc.). A value of 0 is only relevant for style *bond*; see details below. For numeric types, a wildcard asterisk can be used in place of or in conjunction with the *type* argument to specify a range of types. This takes the form "*" or "*n" or "m*" or "m*n". If *N* is the number of types, then an asterisk with no numeric values means all types from 0 to *N*. A leading asterisk means all types from 0 to n (inclusive). A trailing asterisk means all types from m to N (inclusive). A middle asterisk means all types from m to n (inclusive). Note that it is fine to include a type of 0 for non-bond styles; it will simply be ignored.

For style *multi* all bond, angle, dihedral, and improper interactions of any type, involving atoms in the group, are turned off.

Style *atom* is the same as style *multi* except that in addition, one or more of the atoms involved in the bond, angle, dihedral, or improper interaction must also be of the specified atom type.

For style *bond*, only bonds are candidates for turn-off, and the bond must also be of the specified type. Styles *angle*, *dihedral*, and *improper* are treated similarly.

For style *bond*, you can set the type to 0 to delete bonds that have been previously broken by a bond-breaking potential (which sets the bond type to 0 when a bond is broken); for example, see the *bond_style quartic* command.

For style *stats* no interactions are turned off (or on); the status of all interactions in the specified group is simply reported. This is useful for diagnostic purposes if bonds have been turned off by a bond-breaking potential during a previous run.

The default behavior of the delete_bonds command is to turn off interactions by toggling their type to a negative value, but not to permanently remove the interaction. For example, a bond_type of 2 is set to −2. The neighbor list creation routines will not include such an interaction in their interaction lists. The default is also to not alter the list of 1–2, 1–3, or 1–4 neighbors computed by the *special_bonds* command and used to weight pairwise force and energy calculations. This means that pairwise computations will proceed as if the bond (or angle, etc.) were still turned on.

Several keywords can be appended to the argument list to alter the default behaviors.

The *any* keyword changes the requirement that all atoms in the bond (angle, etc.) must be in the specified group in order to turn off the interaction. Instead, if any of the atoms in the interaction are in the specified group, it will be turned off (or on if the *undo* keyword is used).

The *undo* keyword inverts the delete_bonds command so that the specified bonds, angles, etc. are turned on if they are currently turned off. This means a negative value is toggled to positive. For example, for style *angle*, if *type* is specified as 2, then all angles with current type = −2 are reset to type = 2. Note that the *fix shake* command also sets bond and angle types negative, so this option should not be used on those interactions.

The *remove* keyword is invoked at the end of the delete_bonds operation. It causes turned-off bonds (angles, etc.) to be removed from each atom's data structure and then adjusts the global bond (angle, etc.) counts accordingly. Removal is a permanent change; removed bonds cannot be turned back on via the *undo* keyword. Removal does not alter the pairwise 1–2, 1–3, or 1–4 weighting list.

The *special* keyword is invoked at the end of the delete_bonds operation, after (optional) removal. It re-computes the pairwise 1–2, 1–3, 1–4 weighting list. The weighting list computation treats turned-off bonds the same as turned-on. Thus, turned-off bonds must be removed if you wish to change the weighting list.

Note that the choice of *remove* and *special* options affects how 1–2, 1–3, 1–4 pairwise interactions will be computed across bonds that have been modified by the delete_bonds command.

1.21.4 Restrictions

This command requires inter-processor communication to acquire ghost atoms, to coordinate the deleting of bonds, angles, etc. between atoms shared by multiple processors. This means that your system must be ready to perform a simulation before using this command (force fields setup, atom masses set, etc.). Just as would be needed to run dynamics, the force field you define should define a cutoff (e.g., through a *pair_style* command) which is long enough for a processor to acquire the ghost atoms its needs to compute bond, angle, etc. interactions.

If deleted bonds (or angles, etc.) are removed but the 1–2, 1–3, and 1–4 weighting list is not recomputed, this can cause a later *fix shake* command to fail due to an atom's bonds being inconsistent with the weighting list. This should only happen if the group used in the fix command includes both atoms in the bond, in which case you probably should be recomputing the weighting list.

1.21.5 Related commands

neigh_modify exclude, *special_bonds*, *fix shake*

1.21.6 Default

none

1.22 dielectric command

1.22.1 Syntax

```
dielectric value
```

- value = dielectric constant

1.22.2 Examples

```
dielectric 2.0
```

1.22.3 Description

Set the dielectric constant for Coulombic interactions (pairwise and long-range) to this value. The constant is unitless, since it is used to reduce the strength of the interactions. The value is used in the denominator of the formulas for Coulombic interactions (e.g., a value of 4.0 reduces the Coulombic interactions to 25% of their default strength). See the *pair_style* command for more details.

1.22.4 Restrictions

none

1.22.5 Related commands

pair_style

1.22.6 Default

```
dielectric 1.0
```

1.23 dihedral_coeff command

1.23.1 Syntax

```
dihedral_coeff N args
```

- N = numeric dihedral type (see asterisk form below) or alphanumeric type label
- args = coefficients for one or more dihedral types

1.23.2 Examples

```
dihedral_coeff 1 80.0 1 3
dihedral_coeff * 80.0 1 3 0.5
dihedral_coeff 2* 80.0 1 3 0.5

labelmap dihedral 1 backbone
dihedral_coeff backbone 80.0 1 3
```

1.23.3 Description

Specify the dihedral force field coefficients for one or more dihedral types. The number and meaning of the coefficients depends on the dihedral style. Dihedral coefficients can also be set in the data file read by the *read_data* command or in a restart file.

N can be specified in one of two ways. An explicit numeric value can be used, as in the first example above. Or *N* can be an alphanumeric type label, which is a string defined by the *labelmap* command or in a corresponding section of a data file read by the *read_data* command.

For numeric values only, a wild-card asterisk can be used to set the coefficients for multiple dihedral types. This takes the form "*" or "*n" or "n*" or "m*n". If *N* is the number of dihedral types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

Note that using a dihedral_coeff command can override a previous setting for the same dihedral type. For example, these commands set the coeffs for all dihedral types, then overwrite the coeffs for just dihedral type 2:

```
dihedral_coeff * 80.0 1 3
dihedral_coeff 2 200.0 1 3
```

A line in a data file that specifies dihedral coefficients uses the exact same format as the arguments of the dihedral_coeff command in an input script, except that wild-card asterisks should not be used since coefficients for all *N* types must be listed in the file. For example, under the "Dihedral Coeffs" section of a data file, the line that corresponds to the first example above would be listed as

```
1 80.0 1 3
```

The *dihedral_style class2* is an exception to this rule, in that an additional argument is used in the input script to allow specification of the cross-term coefficients. See its doc page for details.

Note

When comparing the formulas and coefficients for various LAMMPS dihedral styles with dihedral equations defined by other force fields, note that some force field implementations divide/multiply the energy prefactor K by the multiple number of torsions that contain the J–K bond in an I-J-K-L torsion. LAMMPS does not do this (i.e., the listed dihedral equation applies to each individual dihedral). Thus, you need to define K appropriately to account for this difference, if necessary.

The list of all dihedral styles defined in LAMMPS is given on the *dihedral_style* doc page. They are also listed in more compact form on the *Commands dihedral* doc page.

On either of those pages, click on the style to display the formula it computes and its coefficients as specified by the associated dihedral_coeff command.

1.23.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

A dihedral style must be defined before any dihedral coefficients are set, either in the input script or in a data file.

1.23.5 Related commands

dihedral_style

1.23.6 Default

none

1.24 dihedral_style command

1.24.1 Syntax

```
dihedral_style style
```

- style = *none* or *zero* or *hybrid* or *charmm* or *charmmfsw* or *class2* or *cosine/shift/exp* or *cosine/squared/restricted* or *fourier* or *harmonic* or *helix* or *lepton* or *multi/harmonic* or *nharmonic* or *opls* or *spherical* or *table* or *table/cut*

1.24.2 Examples

```
dihedral_style harmonic
dihedral_style multi/harmonic
dihedral_style hybrid harmonic charmm
```

1.24.3 Description

Set the formula(s) LAMMPS uses to compute dihedral interactions between quadruplets of atoms, which remain in force for the duration of the simulation. The list of dihedral quadruplets is read in by a *read_data* or *read_restart* command from a data or restart file.

Hybrid models where dihedrals are computed using different dihedral potentials can be setup using the *hybrid* dihedral style.

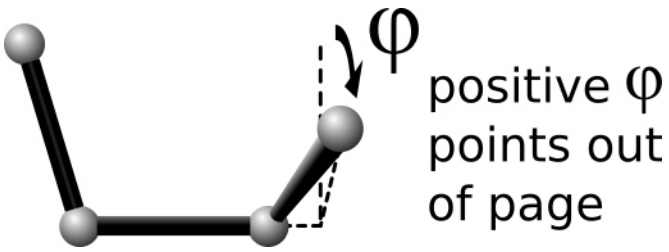
The coefficients associated with a dihedral style can be specified in a data or restart file or via the *dihedral_coeff* command.

All dihedral potentials store their coefficient data in binary restart files which means dihedral_style and *dihedral_coeff* commands do not need to be re-specified in an input script that restarts a simulation. See the *read_restart* command for details on how to do this. The one exception is that dihedral_style *hybrid* only stores the list of sub-styles in the restart file; dihedral coefficients need to be re-specified.

Note

When both a dihedral and pair style is defined, the *special_bonds* command often needs to be used to turn off (or weight) the pairwise interaction that would otherwise exist between four bonded atoms.

In the formulas listed for each dihedral style, *phi* is the torsional angle defined by the quadruplet of atoms. This angle has a sign convention as shown in this diagram:



where the I, J, K, L ordering of the four atoms that define the dihedral is from left to right.

This sign convention effects several of the dihedral styles listed below (e.g., charmm, helix) in the sense that the energy formula depends on the sign of phi, which may be reflected in the value of the coefficients you specify.

Note

When comparing the formulas and coefficients for various LAMMPS dihedral styles with dihedral equations defined by other force fields, note that some force field implementations divide/multiply the energy prefactor K by the multiple number of torsions that contain the J–K bond in an I–J–K–L torsion. LAMMPS does not do this (i.e., the listed dihedral equation applies to each individual dihedral). Thus, you need to define K appropriately via the *dihedral_coeff* command to account for this difference if necessary.

Here is an alphabetic list of dihedral styles defined in LAMMPS. Click on the style to display the formula it computes and coefficients specified by the associated *dihedral_coeff* command.

Click on the style to display the formula it computes, any additional arguments specified in the dihedral_style command, and coefficients specified by the associated *dihedral_coeff* command.

There are also additional accelerated pair styles included in the LAMMPS distribution for faster performance on CPUs, GPUs, and KNLs. The individual style names on the *Commands dihedral* page are followed by one or more of (g,i,k,o,t) to indicate which accelerated styles exist.

- *none* - turn off dihedral interactions
- *zero* - topology but no interactions
- *hybrid* - define multiple styles of dihedral interactions
- *charmm* - CHARMM dihedral
- *charmmfsw* - CHARMM dihedral with force switching
- *class2* - COMPASS (class 2) dihedral
- *cosine/shift/exp* - dihedral with exponential in spring constant
- *cosine/squared/restricted* - squared cosine dihedral with restricted term
- *fourier* - dihedral with multiple cosine terms
- *harmonic* - harmonic dihedral
- *helix* - helix dihedral
- *lepton* - dihedral potential from evaluating a string
- *multi/harmonic* - dihedral with 5 harmonic terms
- *nharmonic* - same as multi-harmonic with N terms
- *opls* - OPLS dihedral
- *quadratic* - dihedral with quadratic term in angle
- *spherical* - dihedral which includes angle terms to avoid singularities
- *table* - tabulated dihedral
- *table/cut* - tabulated dihedral with analytic cutoff

1.24.4 Restrictions

Dihedral styles can only be set for atom styles that allow dihedrals to be defined.

Most dihedral styles are part of the MOLECULE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info. The doc pages for individual dihedral potentials tell if it is part of a package.

1.24.5 Related commands

dihedral_coeff

1.24.6 Default

dihedral_style none

1.25 dihedral_write command

1.25.1 Syntax

```
dihedral_write dtype N file keyword
```

- dtype = dihedral type
- N = # of values
- file = name of file to write values to
- keyword = section name in file for this set of tabulated values

1.25.2 Examples

```
dihedral_write 1  500 table.txt Harmonic_1
dihedral_write 3 1000 table.txt Harmonic_3
```

1.25.3 Description

Added in version 8Feb2023.

Write energy and force values to a file as a function of the dihedral angle for the currently defined dihedral potential. Force in this context means the force with respect to the dihedral angle, not the force on individual atoms. This is useful for plotting the potential function or otherwise debugging its values. The resulting file can also be used as input for use with *dihedral style table*.

If the file already exists, the table of values is appended to the end of the file to allow multiple tables of energy and force to be included in one file. The individual sections may be identified by the *keyword*.

The energy and force values are computed for dihedrals ranging from 0 degrees to 360 degrees for 4 interacting atoms forming an dihedral type dtype, using the appropriate *dihedral_coeff* coefficients. N evenly spaced dihedrals are used. Since 0 and 360 degrees are the same dihedral angle, the latter entry is skipped.

For example, for N = 6, values would be computed at $\phi = 0, 60, 120, 180, 240, 300$.

The file is written in the format used as input for the *dihedral_style table* option with *keyword* as the section name. Each line written to the file lists an index number (1-N), an dihedral angle (in degrees), an energy (in energy units), and a force (in force units per radians^2). In case a new file is created, the first line will be a comment with a "DATE:" and "UNITS:" tag with the current date and *units* settings. For subsequent invocations of the *dihedral_write* command for the same file, data will be appended and the current units settings will be compared to the data from the header, if present. The *dihedral_write* will refuse to add a table to an existing file if the units are not the same.

1.25.4 Restrictions

All force field coefficients for dihedrals and other kinds of interactions must be set before this command can be invoked.

The table of the dihedral energy and force data data is created by using a separate, internally created, new LAMMPS instance with a dummy system of 4 atoms for which the dihedral potential energy is computed after transferring the dihedral style and coefficients and arranging the 4 atoms into the corresponding geometries. The dihedral force is then determined from the potential energies through numerical differentiation. As a consequence of this approach, not all dihedral styles are compatible. The following conditions must be met:

- The dihedral style must be able to write its coefficients to a data file. This condition excludes for example *dihedral style hybrid* and *dihedral style table*.
- The potential function must not have any terms that depend on geometry properties other than the dihedral. This condition excludes for example *dihedral style class2*. Please note that the *write_dihedral* command has no way of checking for this condition. It will check the style name against an internal list of known to be incompatible styles. The resulting tables may be bogus for unlisted dihedral styles if the requirement is not met. It is thus recommended to make careful tests for any created tables.

1.25.5 Related commands

dihedral_style table, *bond_write*, *angle_write*, *dihedral_style*, *dihedral_coeff*

1.25.6 Default

none

1.26 dimension command

1.26.1 Syntax

```
dimension N
```

- N = 2 or 3

1.26.2 Examples

```
dimension 2
```

1.26.3 Description

Set the dimensionality of the simulation. By default LAMMPS runs 3d simulations. To run a 2d simulation, this command should be used prior to setting up a simulation box via the *create_box* or *read_data* commands. Restart files also store this setting.

See the discussion on the *Howto 2d* page for additional instructions on how to run 2d simulations.

Note

Some models in LAMMPS treat particles as finite-size spheres or ellipsoids, as opposed to point particles. In 2d, the particles will still be spheres or ellipsoids, not circular disks or ellipses, meaning their moment of inertia will be the same as in 3d.

1.26.4 Restrictions

This command must be used before the simulation box is defined by a *read_data* or *create_box* command.

1.26.5 Related commands

fix enforce2d

1.26.6 Default

```
dimension 3
```

1.27 displace_atoms command

1.27.1 Syntax

```
displace_atoms group-ID style args keyword value ...
```

- group-ID = ID of group of atoms to displace
- style = *move* or *ramp* or *random* or *rotate*

  move args = delx dely delz
    delx,dely,delz = distance to displace in each dimension (distance units)
    any of delx,dely,delz can be a variable (see below)
  ramp args = ddim dlo dhi dim clo chi
    ddim = x or y or z
    dlo,dhi = displacement distance between dlo and dhi (distance units)
    dim = x or y or z
    clo,chi = lower and upper bound of domain to displace (distance units)
  random args = dx dy dz seed
    dx,dy,dz = random displacement magnitude in each dimension (distance units)
    seed = random # seed (positive integer)
  rotate args = Px Py Pz Rx Ry Rz theta
    Px,Py,Pz = origin point of axis of rotation (distance units)
    Rx,Ry,Rz = axis of rotation vector
    theta = angle of rotation (degrees)

- zero or more keyword/value pairs may be appended

  keyword = units
    units value = box or lattice

1.27.2 Examples

```
displace_atoms top move 0 -5 0 units box
displace_atoms flow ramp x 0.0 5.0 y 2.0 20.5
```

1.27.3 Description

Displace a group of atoms. This can be used to move atoms a large distance before beginning a simulation or to randomize atoms initially on a lattice. For example, in a shear simulation, an initial strain can be imposed on the system. Or two groups of atoms can be brought into closer proximity.

The *move* style displaces the group of atoms by the specified 3d displacement vector. Any of the three quantities defining the vector components can be specified as an equal-style or atom-style *variable*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated, and its value(s) used for the displacement(s). The scale factor implied by the *units* keyword will also be applied to the variable result.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Atom-style variables can specify the same formulas as equal-style variables but can also include per-atom values, such as atom coordinates or per-atom values read from a file. Note that if the variable references other *compute* or *fix* commands, those values must be up-to-date for the current timestep. See the "Variable Accuracy" section of the *variable* doc page for more details.

The *ramp* style displaces atoms a variable amount in one dimension depending on the atom's coordinate in a (possibly) different dimension. For example, the second example command displaces atoms in the x-direction an amount between 0.0 and 5.0 distance units. Each atom's displacement depends on the fractional distance its y coordinate is between 2.0 and 20.5. Atoms with y-coordinates outside those bounds will be moved the minimum (0.0) or maximum (5.0) amount.

The *random* style independently moves each atom in the group by a random displacement, uniformly sampled from a value between $-dx$ and $+dx$ in the x dimension, and similarly for y and z. Random numbers are used in such a way that the displacement of a particular atom is the same, regardless of how many processors are being used.

The *rotate* style rotates each atom in the group by the angle *theta* around a rotation axis $R = (R_x, R_y, R_z)$ that goes through a point $P = (P_x, P_y, P_z)$. The direction of rotation for the atoms around the rotation axis is consistent with the right-hand rule: if your right-hand thumb points along R, then your fingers wrap around the axis in the direction of positive theta.

If the defined *atom_style* assigns an orientation to each atom (*atom styles* ellipsoid, line, tri, body), then that property is also updated appropriately to correspond to the atom's rotation.

Distance units for displacements and the origin point of the *rotate* style are determined by the setting of *box* or *lattice* for the *units* keyword. *Box* means distance units as defined by the *units* command (e.g., Å for *real* or *metal* units). *Lattice* means distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacing.

Note

Care should be taken not to move atoms on top of other atoms. After the move, atoms are remapped into the periodic simulation box if needed, and any shrink-wrap boundary conditions (see the *boundary* command) are enforced which may change the box size. Other than this effect, this command does not change the size or shape of the simulation box. See the *change_box* command if that effect is desired.

Note

Atoms can be moved arbitrarily long distances by this command. If the simulation box is non-periodic and shrink-wrapped (see the *boundary* command), this can change its size or shape. This is not a problem, except that the mapping of processors to the simulation box is not changed by this command from its initial 3d configuration; see the *processors* command. Thus, if the box size/shape changes dramatically, the mapping of processors to the simulation box may not end up as optimal as the initial mapping attempted to be.

1.27.4 Restrictions

For a 2d simulation, only rotations around the a vector parallel to the z-axis are allowed.

1.27.5 Related commands

lattice, *change_box*, *fix move*

1.27.6 Default

The option defaults are units = lattice.

1.28 dynamical_matrix command

Accelerator Variant: dynamical_matrix/kk

1.28.1 Syntax

```
dynamical_matrix group-ID style gamma args keyword value ...
```

- group-ID = ID of group of atoms to displace
- style = *regular* or *eskm*
- gamma = finite different displacement length (distance units)
- one or more keyword/arg pairs may be appended

  keyword = file or binary
    file name = name of output file for the dynamical matrix
    binary arg = yes or no or gzip

1.28.2 Examples

```
dynamical_matrix 1 regular 0.000001
dynamical_matrix 1 eskm 0.000001
dynamical_matrix 3 regular 0.00004 file dynmat.dat
dynamical_matrix 5 eskm 0.00000001 file dynamical.dat binary yes
```

1.28.3 Description

Calculate the dynamical matrix by finite difference of the selected group,

$$D = \frac{\Phi_{ij}^{\alpha\beta}}{\sqrt{M_i M_j}}$$

where D is the dynamical matrix and Φ is the force constant matrix defined by

$$\Phi_{ij}^{\alpha\beta} = \frac{\partial^2 U}{\partial x_{i,\alpha} \partial x_{j,\beta}}$$

The output for the dynamical matrix is printed three elements at a time. The three elements are the three β elements for a respective i/α/j combination. Each line is printed in order of j increasing first, α second, and i last.

If the style eskm is selected, the dynamical matrix will be in units of inverse squared femtoseconds. These units will then conveniently leave frequencies in THz.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

1.28.4 Restrictions

The command collects an array of nine times the number of atoms in a group on every single MPI rank, so the memory requirements can be very significant for large systems.

This command is part of the PHONON package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

1.28.5 Related commands

fix phonon, *fix numdiff*,

compute hma uses an analytic formulation of the Hessian provided by a pair_style's Pair::single_hessian() function, if implemented.

1.28.6 Default

The default settings are file = "dynmat.dyn", binary = no

1.29 echo command

1.29.1 Syntax

```
echo style
```

- style = *none* or *screen* or *log* or *both*

1.29.2 Examples

```
echo both
echo log
```

1.29.3 Description

This command determines whether LAMMPS echoes each input script command to the screen and/or log file as it is read and processed. If an input script has errors, it can be useful to look at echoed output to see the last command processed.

The *command-line switch* -echo can be used in place of this command.

1.29.4 Restrictions

none

1.29.5 Related commands

none

1.29.6 Default

```
echo log
```

1.30 fix command

1.30.1 Syntax

```
fix ID group-ID style args
```

- ID = user-assigned name for the fix
- group-ID = ID of the group of atoms to apply the fix to
- style = one of a long list of possible style names (see below)
- args = arguments used by a particular style

1.30.2 Examples

```
fix 1 all nve
fix 3 all nvt temp 300.0 300.0 0.01
fix mine top setforce 0.0 NULL 0.0
```

1.30.3 Description

Set a fix that will be applied to a group of atoms. In LAMMPS, a "fix" is any operation that is applied to the system during timestepping or minimization. Examples include updating of atom positions and velocities due to time integration, controlling temperature, applying constraint forces to atoms, enforcing boundary conditions, computing diagnostics, etc. There are hundreds of fixes defined in LAMMPS and new ones can be added; see the *Modify* page for details.

Fixes perform their operations at different stages of the timestep. If two or more fixes operate at the same stage of the timestep, they are invoked in the order they were specified in the input script.

The ID of a fix can only contain alphanumeric characters and underscores.

Fixes can be deleted with the *unfix* command.

Note

The *unfix* command is the only way to turn off a fix; simply specifying a new fix with a similar style will not turn off the first one. This is especially important to realize for integration fixes. For example, using a *fix nve* command for a second run after using a *fix nvt* command for the first run will not cancel out the NVT time integration invoked by the "fix nvt" command. Thus, two time integrators would be in place!

If you specify a new fix with the same ID and style as an existing fix, the old fix is deleted and the new one is created (presumably with new settings). This is the same as if an "unfix" command were first performed on the old fix, except that the new fix is kept in the same order relative to the existing fixes as the old one originally was. Note that this operation also wipes out any additional changes made to the old fix via the *fix_modify* command.

The *fix modify* command allows settings for some fixes to be reset. See the page for individual fixes for details.

Some fixes store an internal "state" which is written to binary restart files via the *restart* or *write_restart* commands. This allows the fix to continue on with its calculations in a restarted simulation. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file. See the doc pages for individual fixes for info on which ones can be restarted.

Some fixes calculate and store any of four *styles* of quantities: global, per-atom, local, or per-grid.

A global quantity is one or more system-wide values, e.g. the energy of a wall interacting with particles. A per-atom quantity is one or more values per atom, e.g. the original coordinates of each atom at time 0. Per-atom values are set to 0.0 for atoms not in the specified fix group. Local quantities are calculated by each processor based on the atoms it owns, but there may be zero or more per atom, e.g. values for each bond. Per-grid quantities are calculated on a regular 2d or 3d grid which overlays a 2d or 3d simulation domain. The grid points and the data they store are distributed across processors; each processor owns the grid points which fall within its subdomain.

As a general rule of thumb, fixes that produce per-atom quantities have the word "atom" at the end of their style, e.g. *ave/atom*. Fixes that produce local quantities have the word "local" at the end of their style, e.g. *store/local*. Fixes that produce per-grid quantities have the word "grid" at the end of their style, e.g. *ave/grid*.

Global, per-atom, local, and per-grid quantities can also be of three *kinds*: a single scalar value (global only), a vector of values, or a 2d array of values. For per-atom, local, and per-grid quantities, a "vector" means a single value for each atom, each local entity (e.g. bond), or grid cell. Likewise an "array", means multiple values for each atom, each local entity, or each grid cell.

Note that a single fix can produce any combination of global, per-atom, local, or per-grid values. Likewise it can produce any combination of scalar, vector, or array output for each style. The exception is that for per-atom, local, and per-grid output, either a vector or array can be produced, but not both. The doc page for each fix explains the values it produces, if any.

When a fix output is accessed by another input script command it is referenced via the following bracket notation, where ID is the ID of the fix:

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

In other words, using one bracket reduces the dimension of the quantity once (vector → scalar, array → vector). Using two brackets reduces the dimension twice (array → scalar). Thus, for example, a command that uses global scalar fix values as input can also process elements of a vector or array. Depending on the command, this can either be done directly using the syntax in the table, or by first defining a *variable* of the appropriate style to store the quantity, then using the variable as an input to the command.

Note that commands and *variables* which take fix outputs as input typically do not allow for all styles and kinds of data (e.g., a command may require global but not per-atom values, or it may require a vector of values, not a scalar). This means there is typically no ambiguity about referring to a fix output as c_ID even if it produces, for example, both a scalar and vector. The doc pages for various commands explain the details, including how any ambiguities are resolved.

In LAMMPS, the values generated by a fix can be used in several ways:

- Global values can be output via the *thermo_style custom* or *fix ave/time* command. Alternatively, the values can be referenced in an *equal-style variable* command.
- Per-atom values can be output via the *dump custom* command, or they can be time-averaged via the *fix ave/atom* command or reduced by the *compute reduce* command. Alternatively, per-atom values can be referenced in an *atom-style variable*.

- Local values can be reduced by the *compute reduce* command or histogrammed by the *fix ave/histo* command. They can also be output by the *dump local* command.

See the *Howto output* page for a summary of various LAMMPS output options, many of which involve fixes.

The results of fixes that calculate global quantities can be either "intensive" or "extensive" values. Intensive means the value is independent of the number of atoms in the simulation (e.g., temperature). Extensive means the value scales with the number of atoms in the simulation (e.g., total rotational kinetic energy). *Thermodynamic output* will normalize extensive values by the number of atoms in the system, depending on the "thermo_modify norm" setting. It will not normalize intensive values. If a fix value is accessed in another way (e.g., by a *variable*), you may want to know whether it is an intensive or extensive value. See the page for individual fix styles for further info.

Each fix style has its own page that describes its arguments and what it does, as listed below. Here is an alphabetical list of fix styles available in LAMMPS. They are also listed in more compact form on the *Commands fix* doc page.

There are also additional accelerated fix styles included in the LAMMPS distribution for faster performance on CPUs, GPUs, and KNLs. The individual style names on the *Commands fix* doc page are followed by one or more of (g,i,k,o,t) to indicate which accelerated styles exist.

- *accelerate/cos* - apply cosine-shaped acceleration to atoms
- *acks2/reaxff* - apply ACKS2 charge equilibration
- *adapt* - change a simulation parameter over time
- *adapt/fep* - enhanced version of fix adapt
- *addforce* - add a force to each atom
- *add/heat* - add a heat flux to each atom
- *addtorque* - add a torque to a group of atoms
- *alchemy* - perform an "alchemical transformation" between two partitions
- *amoeba/bitorsion* - torsion/torsion terms in AMOEBA force field
- *amoeba/pitorsion* - 6-body terms in AMOEBA force field
- *append/atoms* - append atoms to a running simulation
- *atc* - initiates a coupled MD/FE simulation
- *atom/swap* - Monte Carlo atom type swapping
- *ave/atom* - compute per-atom time-averaged quantities
- *ave/chunk* - compute per-chunk time-averaged quantities
- *ave/correlate* - compute/output time correlations
- *ave/correlate/long* - alternative to *ave/correlate* that allows efficient calculation over long time windows
- *ave/grid* - compute per-grid time-averaged quantities
- *ave/histo* - compute/output time-averaged histograms
- *ave/histo/weight* - weighted version of fix ave/histo
- *ave/time* - compute/output global time-averaged quantities
- *aveforce* - add an averaged force to each atom
- *balance* - perform dynamic load-balancing
- *brownian* - overdamped translational brownian motion

- *brownian/asphere* - overdamped translational and rotational brownian motion for ellipsoids
- *brownian/sphere* - overdamped translational and rotational brownian motion for spheres
- *bocs* - NPT style time integration with pressure correction
- *bond/break* - break bonds on the fly
- *bond/create* - create bonds on the fly
- *bond/create/angle* - create bonds on the fly with angle constraints
- *bond/react* - apply topology changes to model reactions
- *bond/swap* - Monte Carlo bond swapping
- *box/relax* - relax box size during energy minimization
- *charge/regulation* - Monte Carlo sampling of charge regulation
- *cmap* - CMAP torsion/torsion terms in CHARMM force field
- *colvars* - interface to the collective variables "Colvars" library
- *controller* - apply control loop feedback mechanism
- *damping/cundall* - Cundall non-viscous damping for granular simulations
- *deform* - change the simulation box size/shape
- *deform/pressure* - change the simulation box size/shape with additional loading conditions
- *deposit* - add new atoms above a surface
- *dpd/energy* - constant energy dissipative particle dynamics
- *drag* - drag atoms towards a defined coordinate
- *drude* - part of Drude oscillator polarization model
- *drude/transform/direct* - part of Drude oscillator polarization model
- *drude/transform/inverse* - part of Drude oscillator polarization model
- *dt/reset* - reset the timestep based on velocity, forces
- *edpd/source* - add heat source to eDPD simulations
- *efield* - impose electric field on system
- *efield/tip4p* - impose electric field on system with TIP4P molecules
- *ehex* - enhanced heat exchange algorithm
- *electrode/conp* - impose electric potential
- *electrode/conq* - impose total electric charge
- *electrode/thermo* - apply thermo-potentiostat
- *electron/stopping* - electronic stopping power as a friction force
- *electron/stopping/fit* - electronic stopping power as a friction force
- *enforce2d* - zero out z-dimension velocity and force
- *eos/cv* - applies a mesoparticle equation of state to relate the particle internal energy to the particle internal temperature
- *eos/table* - applies a tabulated mesoparticle equation of state to relate the particle internal energy to the particle internal temperature

- *eos/table/rx* - applies a tabulated mesoparticle equation of state to relate the concentration-dependent particle internal energy to the particle internal temperature
- *evaporate* - remove atoms from simulation periodically
- *external* - callback to an external driver program
- *ffl* - apply a Fast-Forward Langevin equation thermostat
- *filter/corotate* - implement corotation filter to allow larger timesteps with r-RESPA
- *flow/gauss* - Gaussian dynamics for constant mass flux
- *freeze* - freeze atoms in a granular simulation
- *gcmc* - grand canonical insertions/deletions
- *gld* - generalized Langevin dynamics integrator
- *gle* - generalized Langevin equation thermostat
- *gravity* - add gravity to atoms in a granular simulation
- *grem* - implements the generalized replica exchange method
- *halt* - terminate a dynamics run or minimization
- *heat* - add/subtract momentum-conserving heat
- *heat/flow* - plain time integration of heat flow with per-atom temperature updates
- *hyper/global* - global hyperdynamics
- *hyper/local* - local hyperdynamics
- *imd* - implements the "Interactive MD" (IMD) protocol
- *indent* - impose force due to an indenter
- *ipi* - enable LAMMPS to run as a client for i-PI path-integral simulations
- *langevin* - Langevin temperature control
- *langevin/drude* - Langevin temperature control of Drude oscillators
- *langevin/eff* - Langevin temperature control for the electron force field model
- *langevin/spin* - Langevin temperature control for a spin or spin-lattice system
- *lb/fluid* - lattice-Boltzmann fluid on a uniform mesh
- *lb/momentum* - *fix momentum* replacement for use with a lattice-Boltzmann fluid
- *lb/viscous* - *fix viscous* replacement for use with a lattice-Boltzmann fluid
- *lineforce* - constrain atoms to move in a line
- *manifoldforce* - restrain atoms to a manifold during minimization
- *mdi/qm* - LAMMPS operates as a client for a quantum code via the MolSSI Driver Interface (MDI)
- *mdi/qmmm* - LAMMPS operates as client for QM/MM simulation with a quantum code via the MolSSI Driver Interface (MDI)
- *meso/move* - move mesoscopic SPH/SDPD particles in a prescribed fashion
- *mol/swap* - Monte Carlo atom type swapping with a molecule
- *momentum* - zero the linear and/or angular momentum of a group of atoms
- *momentum/chunk* - zero the linear and/or angular momentum of a chunk of atoms

- *move* - move atoms in a prescribed fashion
- *msst* - multi-scale shock technique (MSST) integration
- *mvv/dpd* - DPD using the modified velocity-Verlet integration algorithm
- *mvv/edpd* - constant energy DPD using the modified velocity-Verlet algorithm
- *mvv/tdpd* - constant temperature DPD using the modified velocity-Verlet algorithm
- *neb* - nudged elastic band (NEB) spring forces
- *neb/spin* - nudged elastic band (NEB) spring forces for spins
- *nonaffine/displacement* - calculate nonaffine displacement of atoms
- *nph* - constant NPH time integration via Nose/Hoover
- *nph/asphere* - NPH for aspherical particles
- *nph/body* - NPH for body particles
- *nph/eff* - NPH for nuclei and electrons in the electron force field model
- *nph/sphere* - NPH for spherical particles
- *nphug* - constant-stress Hugoniostat integration
- *npt* - constant NPT time integration via Nose/Hoover
- *npt/asphere* - NPT for aspherical particles
- *npt/body* - NPT for body particles
- *npt/cauchy* - NPT with Cauchy stress
- *npt/eff* - NPT for nuclei and electrons in the electron force field model
- *npt/sphere* - NPT for spherical particles
- *npt/uef* - NPT style time integration with diagonal flow
- *numdiff* - numerically approximate atomic forces using finite energy differences
- *numdiff/virial* - numerically approximate virial stress tensor using finite energy differences
- *nve* - constant NVE time integration
- *nve/asphere* - NVE for aspherical particles
- *nve/asphere/noforce* - NVE for aspherical particles without forces
- *nve/awpmd* - NVE for the Antisymmetrized Wave Packet Molecular Dynamics model
- *nve/body* - NVE for body particles
- *nve/dot* - rigid body constant energy time integrator for coarse grain models
- *nve/dotc/langevin* - Langevin style rigid body time integrator for coarse grain models
- *nve/eff* - NVE for nuclei and electrons in the electron force field model
- *nve/limit* - NVE with limited step length
- *nve/line* - NVE for line segments
- *nve/manifold/rattle* - NVE time integration for atoms constrained to a curved surface (manifold)
- *nve/noforce* - NVE without forces (update positions only)
- *nve/sphere* - NVE for spherical particles

- *nve/bpm/sphere* - NVE for spherical particles used in the BPM package
- *nve/spin* - NVE for a spin or spin-lattice system
- *nve/tri* - NVE for triangles
- *nvk* - constant kinetic energy time integration
- *nvt* - NVT time integration via Nose/Hoover
- *nvt/asphere* - NVT for aspherical particles
- *nvt/body* - NVT for body particles
- *nvt/eff* - NVE for nuclei and electrons in the electron force field model
- *nvt/manifold/rattle* - NVT time integration for atoms constrained to a curved surface (manifold)
- *nvt/sllod* - NVT for NEMD with SLLOD equations
- *nvt/sllod/eff* - NVT for NEMD with SLLOD equations for the electron force field model
- *nvt/sphere* - NVT for spherical particles
- *nvt/uef* - NVT style time integration with diagonal flow
- *oneway* - constrain particles on move in one direction
- *orient/bcc* - add grain boundary migration force for BCC
- *orient/fcc* - add grain boundary migration force for FCC
- *orient/eco* - add generalized grain boundary migration force
- *pafi* - constrained force averages on hyper-planes to compute free energies (PAFI)
- *pair* - access per-atom info from pair styles
- *phonon* - calculate dynamical matrix from MD simulations
- *pimd/langevin* - Feynman path-integral molecular dynamics with stochastic thermostat
- *pimd/nvt* - Feynman path-integral molecular dynamics with Nose-Hoover thermostat
- *planeforce* - constrain atoms to move in a plane
- *plumed* - wrapper on PLUMED free energy library
- *poems* - constrain clusters of atoms to move as coupled rigid bodies
- *polarize/bem/gmres* - compute induced charges at the interface between impermeable media with different dielectric constants with generalized minimum residual (GMRES)
- *polarize/bem/icc* - compute induced charges at the interface between impermeable media with different dielectric constants with the successive over-relaxation algorithm
- *polarize/functional* - compute induced charges at the interface between impermeable media with different dielectric constants with the energy variational approach
- *pour* - pour new atoms/molecules into a granular simulation domain
- *precession/spin* - apply a precession torque to each magnetic spin
- *press/berendsen* - pressure control by Berendsen barostat
- *press/langevin* - pressure control by Langevin barostat
- *print* - print text and variables during a simulation
- *propel/self* - model self-propelled particles

- *property/atom* - add customized per-atom values
- *python/invoke* - call a Python function during a simulation
- *python/move* - move particles using a Python function during a simulation run
- *qbmsst* - quantum bath multi-scale shock technique time integrator
- *qeq/comb* - charge equilibration for COMB potential
- *qeq/dynamic* - charge equilibration via dynamic method
- *qeq/fire* - charge equilibration via FIRE minimizer
- *qeq/point* - charge equilibration via point method
- *qeq/reaxff* - charge equilibration for ReaxFF potential
- *qeq/shielded* - charge equilibration via shielded method
- *qeq/slater* - charge equilibration via Slater method
- *qmmm* - functionality to enable a quantum mechanics/molecular mechanics coupling
- *qtb* - implement quantum thermal bath scheme
- *rattle* - RATTLE constraints on bonds and/or angles
- *reaxff/bonds* - write out ReaxFF bond information
- *reaxff/species* - write out ReaxFF molecule information
- *recenter* - constrain the center-of-mass position of a group of atoms
- *restrain* - constrain a bond, angle, dihedral
- *rheo* - integrator for the RHEO package
- *rheo/thermal* - thermal integrator for the RHEO package
- *rheo/oxidation* - create oxidation bonds for the RHEO package
- *rheo/pressure* - pressure calculation for the RHEO package
- *rheo/viscosity* - viscosity calculation for the RHEO package
- *rhok* - add bias potential for long-range ordered systems
- *rigid* - constrain one or more clusters of atoms to move as a rigid body with NVE integration
- *rigid/meso* - constrain clusters of mesoscopic SPH/SDPD particles to move as a rigid body
- *rigid/nph* - constrain one or more clusters of atoms to move as a rigid body with NPH integration
- *rigid/nph/small* - constrain many small clusters of atoms to move as a rigid body with NPH integration
- *rigid/npt* - constrain one or more clusters of atoms to move as a rigid body with NPT integration
- *rigid/npt/small* - constrain many small clusters of atoms to move as a rigid body with NPT integration
- *rigid/nve* - constrain one or more clusters of atoms to move as a rigid body with alternate NVE integration
- *rigid/nve/small* - constrain many small clusters of atoms to move as a rigid body with alternate NVE integration
- *rigid/nvt* - constrain one or more clusters of atoms to move as a rigid body with NVT integration
- *rigid/nvt/small* - constrain many small clusters of atoms to move as a rigid body with NVT integration
- *rigid/small* - constrain many small clusters of atoms to move as a rigid body with NVE integration
- *rx* - solve reaction kinetic ODEs for a defined reaction set

- *saed/vtk* - time-average the intensities from *compute saed*
- *setforce* - set the force on each atom
- *setforce/spin* - set magnetic precession vectors on each atom
- *sgcmc* - fix for hybrid semi-grand canonical MD/MC simulations
- *shake* - SHAKE constraints on bonds and/or angles
- *shardlow* - integration of DPD equations of motion using the Shardlow splitting
- *smd* - applied a steered MD force to a group
- *smd/adjust_dt* - calculate a new stable time increment for use with SMD integrators
- *smd/integrate_tlsph* - explicit time integration with total Lagrangian SPH pair style
- *smd/integrate_ulsph* - explicit time integration with updated Lagrangian SPH pair style
- *smd/move_tri_surf* - update position and velocity near rigid surfaces using SPH integrators
- *smd/setvel* - sets each velocity component, ignoring forces, for Smooth Mach Dynamics
- *smd/wall_surface* - create a rigid wall with a triangulated surface for use in Smooth Mach Dynamics
- *sph* - time integration for SPH/DPDE particles
- *sph/stationary* - update energy and density but not position or velocity in Smooth Particle Hydrodynamics
- *spring* - apply harmonic spring force to group of atoms
- *spring/chunk* - apply harmonic spring force to each chunk of atoms
- *spring/rg* - spring on radius of gyration of group of atoms
- *spring/self* - spring from each atom to its origin
- *srd* - stochastic rotation dynamics (SRD)
- *store/force* - store force on each atom
- *store/state* - store attributes for each atom
- *tdpd/source* - add external concentration source
- *temp/berendsen* - temperature control by Berendsen thermostat
- *temp/csld* - canonical sampling thermostat with Langevin dynamics
- *temp/csvr* - canonical sampling thermostat with Hamiltonian dynamics
- *temp/rescale* - temperature control by velocity rescaling
- *temp/rescale/eff* - temperature control by velocity rescaling in the electron force field model
- *tfmc* - perform force-bias Monte Carlo with time-stamped method
- *tgnvt/drude* - NVT time integration for Drude polarizable model via temperature-grouped Nose-Hoover
- *tgnpt/drude* - NPT time integration for Drude polarizable model via temperature-grouped Nose-Hoover
- *thermal/conductivity* - Mueller-Plathe kinetic energy exchange for thermal conductivity calculation
- *ti/spring* - perform thermodynamic integration between a solid and an Einstein crystal
- *tmd* - guide a group of atoms to a new configuration
- *ttm* - two-temperature model for electronic/atomic coupling (replicated grid)
- *ttm/grid* - two-temperature model for electronic/atomic coupling (distributed grid)

- *ttm/mod* - enhanced two-temperature model with additional options
- *tune/kspace* - auto-tune *k*-space parameters
- *vector* - accumulate a global vector every *N* timesteps
- *viscosity* - Mueller-Plathe momentum exchange for viscosity calculation
- *viscous* - viscous damping for granular simulations
- *viscous/sphere* - viscous damping on angular velocity for granular simulations
- *wall/body/polygon* - time integration for body particles of style *rounded/polygon*
- *wall/body/polyhedron* - time integration for body particles of style *rounded/polyhedron*
- *wall/colloid* - Lennard-Jones wall interacting with finite-size particles
- *wall/ees* - wall for ellipsoidal particles
- *wall/flow* - flow boundary conditions
- *wall/gran* - frictional wall(s) for granular simulations
- *wall/gran/region* - *fix wall/region* equivalent for use with granular particles
- *wall/harmonic* - harmonic spring wall
- *wall/lj1043* - Lennard-Jones 10–4–3 wall
- *wall/lj126* - Lennard-Jones 12–6 wall
- *wall/lj93* - Lennard-Jones 9–3 wall
- *wall/lepton* - Custom Lepton expression wall
- *wall/morse* - Morse potential wall
- *wall/piston* - moving reflective piston wall
- *wall/reflect* - reflecting wall(s)
- *wall/reflect/stochastic* - reflecting wall(s) with finite temperature
- *wall/region* - use region surface as wall
- *wall/region/ees* - use region surface as wall for ellipsoidal particles
- *wall/srd* - slip/no-slip wall for SRD particles
- *wall/table* - Tabulated potential wall wall
- *widom* - Widom insertions of atoms or molecules

1.30.4 Restrictions

Some fix styles are part of specific packages. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info. The doc pages for individual fixes tell if it is part of a package.

1.30.5 Related commands

unfix, *fix_modify*

1.30.6 Default

none

1.31 fix_modify command

1.31.1 Syntax

```
fix_modify fix-ID keyword value ...
```

- fix-ID = ID of the fix to modify
- one or more keyword/value pairs may be appended
- keyword = *bodyforces* or *colname* or *dynamic/dof* or *energy* or *press* or *respa* or *temp* or *virial*

```
bodyforces value = early or late
  early/late = compute rigid-body forces/torques early or late in the timestep
colname values =  ID string
  string = new column header name
  ID = integer from 1 to N, or integer from -1 to -N, where N = # of quantities being output
       or a fix output property keyword or reference to compute, fix, property or variable.
dynamic/dof value = yes or no
  yes/no = do or do not re-compute the number of degrees of freedom (DOF) contributing to the
 ↪temperature
energy value = yes or no
press value = compute ID that calculates a pressure
respa value = 1 to max respa level or 0 (for outermost level)
temp value = compute ID that calculates a temperature
virial value = yes or no
```

1.31.2 Examples

```
fix_modify 3 temp myTemp press myPress
fix_modify 1 energy yes
fix_modify tether respa 2
fix_modify ave colname c_thermo_press Pressure colname 1 Temperature
```

1.31.3 Description

Modify one or more parameters of a previously defined fix. Only specific fix styles support specific parameters. See the doc pages for individual fix commands for info on which ones support which fix_modify parameters.

The *temp* keyword is used to determine how a fix computes temperature. The specified compute ID must have been previously defined by the user via the *compute* command and it must be a style of compute that calculates a temperature. All fixes that compute temperatures define their own compute by default, as described in their documentation. Thus this option allows the user to override the default method for computing T.

The *press* keyword is used to determine how a fix computes pressure. The specified compute ID must have been previously defined by the user via the *compute* command and it must be a style of compute that calculates a pressure. All fixes that compute pressures define their own compute by default, as described in their documentation. Thus this option allows the user to override the default method for computing P.

The *energy* keyword can be used with fixes that support it, which is explained at the bottom of their doc page. *Energy yes* will add a contribution to the potential energy of the system. More specifically, the fix's global or per-atom energy is included in the calculation performed by the *compute pe* or *compute pe/atom* commands. The former is what is used the *thermo_style* command for output of any quantity that includes the global potential energy of the system. Note that the *compute pe* and *compute pe/atom* commands also have an option to include or exclude the contribution from fixes. For fixes that tally a global energy, it can also be printed with thermodynamic output by using the keyword f_ID in the thermo_style custom command, where ID is the fix-ID of the appropriate fix.

Note

If you are performing an *energy minimization* with one of these fixes and want the energy and forces it produces to be part of the optimization criteria, you must specify the *energy yes* setting.

Note

For most fixes that support the *energy* keyword, the default setting is *no*. For a few it is *yes*, when a user would expect that to be the case. The page of each fix gives the default.

The *virial* keyword can be used with fixes that support it, which is explained at the bottom of their doc page. *Virial yes* will add a contribution to the virial of the system. More specifically, the fix's global or per-atom virial is included in the calculation performed by the *compute pressure* or *compute stress/atom* commands. The former is what is used the *thermo_style* command for output of any quantity that includes the global pressure of the system. Note that the *compute pressure* and *compute stress/atom* commands also have an option to include or exclude the contribution from fixes.

Note

If you are performing an *energy minimization* with *box relaxation* and one of these fixes and want the virial contribution of the fix to be part of the optimization criteria, you must specify the *virial yes* setting.

Note

For most fixes that support the *virial* keyword, the default setting is *no*. For a few it is *yes*, when a user would expect that to be the case. The page of each fix gives the default.

For fixes that set or modify forces, it may be possible to select at which *r-RESPA* level the fix operates via the *respa* keyword. The RESPA level at which the fix is active can be selected. This is a number ranging from 1 to the number of levels. If the RESPA level is larger than the current maximum, the outermost level will be used, which is also the default setting. This default can be restored using a value of *0* for the RESPA level. The affected fix has to be enabled to support this feature; if not, *fix_modify* will report an error. Active fixes with a custom RESPA level setting are reported with their specified level at the beginning of a r-RESPA run.

The *dynamic/dof* keyword determines whether the number of atoms N in the fix group and their associated degrees of freedom are re-computed each time a temperature is computed. Only fix styles that calculate their own internal temperature use this option. Currently this is only the *fix rigid/nvt/small* and *fix rigid/npt/small* commands for the purpose of thermostatting rigid body translation and rotation. By default, N and their DOF are assumed to be constant. If you are adding atoms or molecules to the system (see the *fix pour*, *fix deposit*, and *fix gcmc* commands) or expect atoms or molecules to be lost (e.g. due to exiting the simulation box or via *fix evaporate*), then this option should be used to ensure the temperature is correctly normalized.

Note

Other thermostatting fixes, such as *fix nvt*, do not use the *dynamic/dof* keyword because they use a temperature compute to calculate temperature. See the *compute_modify dynamic/dof* command for a similar way to ensure correct temperature normalization for those thermostats.

The *bodyforces* keyword determines whether the forces and torques acting on rigid bodies are computed *early* at the post-force stage of each timestep (right after per-atom forces have been computed and communicated among processors), or *late* at the final-integrate stage of each timestep (after any other fixes have finished their post-force tasks). Only the rigid-body integration fixes use this option, which includes *fix rigid* and *fix rigid/small*, and their variants, and also *fix poems*.

The default is *late*. If there are other fixes that add forces to individual atoms, then the rigid-body constraints will include these forces when time-integrating the rigid bodies. If *early* is specified, then new fixes can be written that use or modify the per-body force and torque, before time-integration of the rigid bodies occurs. Note however this has the side effect, that fixes such as *fix addforce*, *fix setforce*, *fix spring*, which add forces to individual atoms will have no effect on the motion of the rigid bodies if they are specified in the input script after the fix rigid command. LAMMPS will give a warning if that is the case.

The *colname* keyword can be used to change the default header keywords in output files of fix styles that support it: currently only *fix ave/time* is supported. The setting for *ID string* replaces the default text with the provided string. *ID* can be a positive integer when it represents the column number counting from the left, a negative integer when it represents the column number from the right (i.e. -1 is the last column/keyword), or a custom fix output keyword (or compute, fix, property, or variable reference) and then it replaces the string for that specific keyword. The *colname* keyword can be used multiple times. If multiple *colname* settings refer to the same keyword, the last setting has precedence.

1.31.4 Restrictions

none

1.31.5 Related commands

fix, *compute temp*, *compute pressure*, *thermo_style*

1.31.6 Default

The option defaults are temp = ID defined by fix, press = ID defined by fix, energy = no, virial = different for each fix style, respa = 0, bodyforce = late.

1.32 fitpod command

1.32.1 Syntax

```
fitpod Ta_param.pod Ta_data.pod Ta_coefficients.pod
```

- fitpod = style name of this command
- Ta_param.pod = an input file that describes proper orthogonal descriptors (PODs)
- Ta_data.pod = an input file that specifies DFT data used to fit a POD potential
- Ta_coefficients.pod (optional) = an input file that specifies trainable coefficients of a POD potential

1.32.2 Examples

```
fitpod Ta_param.pod Ta_data.pod
fitpod Ta_param.pod Ta_data.pod Ta_coefficients.pod
```

1.32.3 Description

Added in version 22Dec2022.

Fit a machine-learning interatomic potential (ML-IAP) based on proper orthogonal descriptors (POD); please see *(Nguyen and Rohskopf)*, *(Nguyen2023)*, *(Nguyen2024)*, and *(Nguyen and Sema)* for details. The fitted POD potential can be used to run MD simulations via *pair_style pod*.

Two input files are required for this command. The first input file describes a POD potential parameter settings, while the second input file specifies the DFT data used for the fitting procedure. All keywords except *species* have default values. If a keyword is not set in the input file, its default value is used. The table below has one-line descriptions of all the keywords that can be used in the first input file (i.e. Ta_param.pod)

Keyword	Default	Type	Description
species	(none)	STRING	Chemical symbols for all elements in the system and have to match XYZ training files.
pbc	1 1 1	INT	three integer constants specify boundary conditions
rin	0.5	REAL	a real number specifies the inner cut-off radius
rcut	5.0	REAL	a real number specifies the outer cut-off radius
bessel_polynomial_degree	4	INT	the maximum degree of Bessel polynomials
inverse_polynomial_degree	8	INT	the maximum degree of inverse radial basis functions
number_of_environment_clusters	1	INT	the number of clusters for environment-adaptive potentials
number_of_principal_components	2	INT	the number of principal components for dimensionality reduction
onebody	1	BOOL	turns on/off one-body potential
twobody_number_radial_basis_functions	8	INT	number of radial basis functions for two-body potential
threebody_number_radial_basis_functions	6	INT	number of radial basis functions for three-body potential
threebody_angular_degree	5	INT	angular degree for three-body potential
fourbody_number_radial_basis_functions	4	INT	number of radial basis functions for four-body potential
fourbody_angular_degree	3	INT	angular degree for four-body potential
fivebody_number_radial_basis_functions	0	INT	number of radial basis functions for five-body potential
fivebody_angular_degree	0	INT	angular degree for five-body potential
sixbody_number_radial_basis_functions	0	INT	number of radial basis functions for six-body potential
sixbody_angular_degree	0	INT	angular degree for six-body potential
sevenbody_number_radial_basis_functions	0	INT	number of radial basis functions for seven-body potential
sevenbody_angular_degree	0	INT	angular degree for seven-body potential

Note that both the number of radial basis functions and angular degree must decrease as the body order increases. The next table describes all keywords that can be used in the second input file (i.e. $\mathrm{Ta\_data.pod}$ in the example above):

Keyword	Default	Type	Description
file_format	extxyz	STRING	only the extended xyz format (extxyz) is currently supported
file_extension	xyz	STRING	extension of the data files
path_to_training_data_set	(none)	STRING	specifies the path to training data files in double quotes
path_to_test_data_set	""	STRING	specifies the path to test data files in double quotes
path_to_environment_configuration_set	""	STRING	specifies the path to environment configuration files in double quotes
fraction_training_data_set	1.0	REAL	a real number (<= 1.0) specifies the fraction of the training set used to fit POD
randomize_training_data_set	0	BOOL	turns on/off randomization of the training set
fraction_test_data_set	1.0	REAL	a real number (<= 1.0) specifies the fraction of the test set used to validate POD
randomize_test_data_set	0	BOOL	turns on/off randomization of the test set
fitting_weight_energy	100.0	REAL	a real constant specifies the weight for energy in the least-squares fit
fitting_weight_force	1.0	REAL	a real constant specifies the weight for force in the least-squares fit
fitting_regularization_parameter	1.0e-10	REAL	a real constant specifies the regularization parameter in the least-squares fit
error_analysis_for_training_data_set	0	BOOL	turns on/off error analysis for the training data set
error_analysis_for_test_data_set	0	BOOL	turns on/off error analysis for the test data set
basename_for_output_files	pod	STRING	a basename string added to the output files
precision_for_pod_coefficients	8	INT	number of digits after the decimal points for numbers in the coefficient file
group_weights	global	STRING	table uses group weights defined for each group named by filename

All keywords except *path_to_training_data_set* have default values. If a keyword is not set in the input file, its default value is used. After successful training, a number of output files are produced, if enabled:

- <basename>_training_errors.pod reports the errors in energy and forces for the training data set
- <basename>_training_analysis.pod reports detailed errors for all training configurations
- <basename>_test_errors.pod reports errors for the test data set
- <basename>_test_analysis.pod reports detailed errors for all test configurations
- <basename>_coefficients.pod contains the coefficients of the POD potential

After training the POD potential, Ta_param.pod and <basename>_coefficients.pod are the two files needed to use the POD potential in LAMMPS. See *pair_style pod* for using the POD potential. Examples about training and using POD potentials are found in the directory lammps/examples/PACKAGES/pod and the Github repo https://github.com/cesmix-mit/pod-examples.

Loss Function Group Weights

The *group_weights* keyword in the data.pod file is responsible for weighting certain groups of configurations in the loss function. For example:

```
group_weights table
Displaced_A15 100.0 1.0
Displaced_BCC 100.0 1.0
Displaced_FCC 100.0 1.0
Elastic_BCC   100.0 1.0
Elastic_FCC   100.0 1.0
GSF_110       100.0 1.0
GSF_112       100.0 1.0
Liquid        100.0 1.0
Surface       100.0 1.0
Volume_A15    100.0 1.0
Volume_BCC    100.0 1.0
Volume_FCC    100.0 1.0
```

This will apply an energy weight of 100.0 and a force weight of 1.0 for all groups in the Ta example. The groups are named by their respective filename. If certain groups are left out of this table, then the globally defined weights from the fitting_weight_energy and fitting_weight_force keywords will be used.

1.32.4 POD Potential

We consider a multi-element system of N atoms with N_e unique elements. We denote by r_n and Z_n position vector and type of an atom n in the system, respectively. Note that we have $Z_n \in \{1, \ldots, N_e\}$, $R = (r_1, r_2, \ldots, r_N) \in \mathbb{R}^{3N}$, and $Z = (Z_1, Z_2, \ldots, Z_N) \in \mathbb{N}^N$. The total energy of the POD potential is expressed as $E(R, Z) = \sum_{i=1}^{N} E_i(R_i, Z_i)$, where

$$E_i(R_i, Z_i) \;=\; \sum_{m=1}^{M} c_m \mathcal{D}_{im}(R_i, Z_i)$$

Here c_m are trainable coefficients and $\mathcal{D}_{im}(R_i, Z_i)$ are per-atom POD descriptors. Summing the per-atom descriptors over i yields the global descriptors $d_m(R, Z) = \sum_{i=1}^{N} \mathcal{D}_{im}(R_i, Z_i)$. It thus follows that $E(R, Z) = \sum_{m=1}^{M} c_m d_m(R, Z)$.

The per-atom POD descriptors include one, two, three, four, five, six, and seven-body descriptors, which can be specified in the first input file. Furthermore, the per-atom POD descriptors also depend on the number of environment clusters specified in the first input file. Please see *(Nguyen2024)* and *(Nguyen and Sema)* for the detailed description of the per-atom POD descriptors.

1.32.5 Training

A POD potential is trained using the least-squares regression against density functional theory (DFT) data. Let J be the number of training configurations, with N_j being the number of atoms in the j-th configuration. The training configurations are extracted from the extended XYZ files located in a directory (i.e., path_to_training_data_set in the second input file). Let $\{E_j^\star\}_{j=1}^J$ and $\{F_j^\star\}_{j=1}^J$ be the DFT energies and forces for J configurations. Next, we calculate the global descriptors and their derivatives for all training configurations. Let d_{jm}, $1 \le m \le M$, be the global descriptors associated with the j-th configuration, where M is the number of global descriptors. We then form a matrix $A \in \mathbb{R}^{J \times M}$ with entries $A_{jm} = d_{jm}/N_j$ for $j = 1, \ldots, J$ and $m = 1, \ldots, M$. Moreover, we form a matrix $B \in \mathbb{R}^{\mathcal{N} \times M}$ by stacking the derivatives of the global descriptors for all training configurations from top to bottom, where $\mathcal{N} = 3\sum_{j=1}^{J} N_j$.

The coefficient vector c of the POD potential is found by solving the following least-squares problem

$$\min_{c \in \mathbb{R}^M} w_E \|Ac - \bar{E}^\star\|^2 + w_F \|Bc + F^\star\|^2 + w_R \|c\|^2,$$

where w_E and w_F are weights for the energy (*fitting_weight_energy*) and force (*fitting_weight_force*), respectively; and w_R is the regularization parameter (*fitting_regularization_parameter*). Here $\bar{E}^\star \in \mathbb{R}^J$ is a vector of with entries $\bar{E}^\star_j = E^\star_j/N_j$ and $F^\star$ is a vector of $\mathcal{N}$ entries obtained by stacking $\{F^\star_j\}^J_{j=1}$ from top to bottom.

1.32.6 Validation

POD potential can be validated on a test dataset in a directory specified by setting path_to_test_data_set in the second input file. It is possible to validate the POD potential after the training is complete. This is done by providing the coefficient file as an input to *fitpod*, for example,

```
fitpod Ta_param.pod Ta_data.pod Ta_coefficients.pod
```

1.32.7 Restrictions

This command is part of the ML-POD package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

1.32.8 Related commands

pair_style pod, *compute pod/atom*, *compute podd/atom*, *compute pod/local*, *compute pod/global*

1.32.9 Default

The keyword defaults are also given in the description of the input files.

(Nguyen and Rohskopf) Nguyen and Rohskopf, Journal of Computational Physics, 480, 112030, (2023).

(Nguyen2023) Nguyen, Physical Review B, 107(14), 144103, (2023).

(Nguyen2024) Nguyen, Journal of Computational Physics, 113102, (2024).

(Nguyen and Sema) Nguyen and Sema, https://arxiv.org/abs/2405.00306, (2024).

1.33 geturl command

1.33.1 Syntax

```
geturl url keyword args ...
```

- url = URL of the file to download
- zero or more keyword argument pairs may be provided
- keyword = *output* or *verify* or *overwrite* or *verbose*

  output filename = write to filename instead of inferring the name from the URL
  verify yes/no = verify SSL certificate and hostname if yes, do not if no
  overwrite yes/no = if yes overwrite the output file in case it exists, do not if no
  verbose yes/no = if yes write verbose debug output from libcurl to screen, do not if no

1.33.2 Examples

```
geturl https://www.ctcms.nist.gov/potentials/Download/1990--Ackland-G-J-Vitek-V--Cu/2/Cu2.eam.fs
geturl https://github.com/lammps/lammps/blob/develop/bench/in.lj output in.bench-lj
```

1.33.3 Description

Added in version 29Aug2024.

Download a file from an URL to the local disk. This is implemented with the libcurl library which supports a large variety of protocols including "http", "https", "ftp", "scp", "sftp", "file". The transfer will only be performed on MPI rank 0.

The *output* keyword can be used to set the filename. By default, the last part of the URL is used.

The *verify* keyword determines whether libcurl will validate the SSL certificate and hostname for encrypted connections. Turning this off may be required when using a proxy or connecting to a server with a self-signed SSL certificate.

The *overwrite* keyword determines whether a file should be overwritten if it already exists. If the argument is *no*, then the download will be skipped if the file exists.

The *verbose* keyword determines whether a detailed protocol of the steps performed by libcurl is written to the screen. Using the argument *yes* can be used to debug connection issues when the *geturl* command does not behave as expected. If the argument is *no*, geturl will operate silently and only report the error status number provided by libcurl, in case of a failure.

1.33.4 Restrictions

This command is part of the EXTRA-COMMAND package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info. It also requires that LAMMPS was built with support for the libcurl library. See the page about *Compiling LAMMPS with libcurl support* for further info. If support for libcurl is not included, using *geturl* will trigger an error.

1.33.5 Related commands

shell

1.33.6 Default

verify = yes, *overwrite* = yes

1.34 group command

1.34.1 Syntax

```
group ID style args
```

- ID = user-defined name of the group
- style = *delete* or *clear* or *empty* or *region* or *type* or *id* or *molecule* or *variable* or *include* or *subtract* or *union* or *intersect* or *dynamic* or *static*

  delete = no args
  clear = no args
  empty = no args
  region args = region-ID
  type or id or molecule
    args = list of one or more atom types (1-Ntypes or type label), atom IDs, or molecule IDs
      any numeric entry in list can be a sequence formatted as A:B or A:B:C where
      A = starting index, B = ending index,
      C = increment between indices, 1 if not specified
    args = logical value
      logical = "<" or "<=" or ">" or ">=" or "==" or "!="
      value = an atom type (1-Ntypes or type label) or atom ID or molecule ID (depending on style)
    args = logical value1 value2
      logical = "<>"
      value1,value2 = atom types or atom IDs or molecule IDs (depending on style)
  variable args = variable-name
  include args = molecule
    molecule = add atoms to group with same molecule ID as atoms already in group
  subtract args = two or more group IDs
  union args = one or more group IDs
  intersect args = two or more group IDs
  dynamic args = parent-ID keyword value ...
    one or more keyword/value pairs may be appended
    keyword = region or var or property or every
      region value = region-ID
      var value = name of variable
      property value = name of custom integer or floating point vector
      every value = N = update group every this many timesteps
  static = no args

1.34.2 Examples

```
group edge region regstrip
group water type 3 4
group water type OW HT
group sub id 10 25 50
group sub id 10 25 50 500:1000
group sub id 100:10000:10
group sub id <= 150
group polyA molecule <> 50 250
group hienergy variable eng
```

(continues on next page)

(continued from previous page)

```
group hienergy include molecule
group boundary subtract all a2 a3
group boundary union lower upper
group boundary intersect upper flow
group boundary delete
group mine dynamic all region myRegion every 100
```

1.34.3 Description

Identify a collection of atoms as belonging to a group. The group ID can then be used in other commands such as *fix*, *compute*, *dump*, or *velocity* to act on those atoms together.

If the group ID already exists, the group command adds the specified atoms to the group.

Note

By default groups are static, meaning the atoms are permanently assigned to the group. For example, if the *region* style is used to assign atoms to a group, the atoms will remain in the group even if they later move out of the region. As explained below, the *dynamic* style can be used to make a group dynamic so that a periodic determination is made as to which atoms are in the group. Since many LAMMPS commands operate on groups of atoms, you should think carefully about whether making a group dynamic makes sense for your model.

A group with the ID *all* is predefined. All atoms belong to this group. This group cannot be deleted, or made dynamic.

The *delete* style removes the named group and un-assigns all atoms that were assigned to that group. Since there is a restriction (see below) that no more than 32 groups can be defined at any time, the *delete* style allows you to remove groups that are no longer needed, so that more can be specified. You cannot delete a group if it has been used to define a current *fix* or *compute* or *dump*.

The *clear* style un-assigns all atoms that were assigned to that group. This may be dangerous to do during a simulation run (e.g., using the *run every* command if a fix or compute or other operation expects the atoms in the group to remain constant), but LAMMPS does not check for this.

The *empty* style creates an empty group, which is useful for commands like *fix gcmc* or with complex scripts that add atoms to a group.

The *region* style puts all atoms in the region volume into the group. Note that this is a static one-time assignment. The atoms remain assigned (or not assigned) to the group even in they later move out of the region volume.

The *type*, *id*, and *molecule* styles put all atoms with the specified atom types, atom IDs, or molecule IDs into the group. These three styles can use arguments specified in one of two formats.

The first format is a list of values (types or IDs). For example, the second command in the examples above puts all atoms of type 3 or 4 into the group named *water*. Each numeric entry in the list can be a colon-separated sequence A:B or A:B:C, as in two of the examples above. A "sequence" generates a sequence of values (types or IDs), with an optional increment. The first example with 500:1000 has the default increment of 1 and would add all atom IDs from 500 to 1000 (inclusive) to the group sub, along with 10, 25, and 50 since they also appear in the list of values. The second example with 100:10000:10 uses an increment of 10 and would thus would add atoms IDs $100, 110, 120, \ldots, 9990, 10000$ to the group sub.

The second format is a *logical* followed by one or two values (type or ID). The 7 valid logicals are listed above. All the logicals except <> take a single argument. The third example above adds all atoms with IDs from 1 to 150 to the group named *sub*. The logical <> means "between" and takes 2 arguments. The fourth example above adds all atoms

belonging to molecules with IDs from 50 to 250 (inclusive) to the group named polyA. For the *type* style, type labels are converted into numeric types before being evaluated.

The *variable* style evaluates a variable to determine which atoms to add to the group. It must be an *atom-style variable* previously defined in the input script. If the variable evaluates to a non-zero value for a particular atom, then that atom is added to the specified group.

Atom-style variables can specify formulas that include thermodynamic quantities, per-atom values such as atom coordinates, or per-atom quantities calculated by computes, fixes, or other variables. They can also include Boolean logic where two numeric values are compared to yield a 1 or 0 (effectively a true or false). Thus, using the *variable* style is a general way to flag specific atoms to include or exclude from a group.

For example, these lines define a variable "eatom" that calculates the potential energy of each atom and includes it in the group if its potential energy is above the threshold value -3.0.

```
compute         1 all pe/atom
compute         2 all reduce sum c_1
thermo_style    custom step temp pe c_2
run             0

variable        eatom atom "c_1 > -3.0"
group           hienergy variable eatom
```

Note that these lines

```
compute         2 all reduce sum c_1
thermo_style    custom step temp pe c_2
run             0
```

are necessary to ensure that the "eatom" variable is current when the group command invokes it. Because the eatom variable computes the per-atom energy via the pe/atom compute, it will only be current if a run has been performed which evaluated pairwise energies, and the pe/atom compute was actually invoked during the run. Printing the thermodynamic info for compute 2 ensures that this is the case, since it sums the pe/atom compute values (in the reduce compute) to output them to the screen. See the "Variable Accuracy" section of the *variable* page for more details on ensuring that variables are current when they are evaluated between runs.

The *include* style with its arg *molecule* adds atoms to a group that have the same molecule ID as atoms already in the group. The molecule ID = 0 is ignored in this operation, since it is assumed to flag isolated atoms that are not part of molecules. An example of where this operation is useful is if the *region* style has been used previously to add atoms to a group that are within a geometric region. If molecules straddle the region boundary, then atoms outside the region that are part of molecules with atoms inside the region will not be in the group. Using the group command a second time with *include molecule* will add those atoms that are outside the region to the group.

Note

The *include molecule* operation is relatively expensive in a parallel sense. This is because it requires communication of relevant molecule IDs between all the processors and each processor to loop over its atoms once per processor, to compare its atoms to the list of molecule IDs from every other processor. Hence it scales as N, rather than N/P as most of the group operations do, where N is the number of atoms, and P is the number of processors.

The *subtract* style takes a list of two or more existing group names as arguments. All atoms that belong to the first group, but not to any of the other groups are added to the specified group.

The *union* style takes a list of one or more existing group names as arguments. All atoms that belong to any of the listed groups are added to the specified group.

The *intersect* style takes a list of two or more existing group names as arguments. Atoms that belong to every one of the listed groups are added to the specified group.

The *dynamic* style flags an existing or new group as dynamic. This means atoms will be (re)assigned to the group periodically as a simulation runs. This is in contrast to static groups where atoms are permanently assigned to the group. The way the assignment occurs is as follows. Only atoms in the group specified as the parent group via the parent-ID are assigned to the dynamic group before the following conditions are applied.

If the *region* keyword is used, atoms not in the specified region are removed from the dynamic group.

If the *var* keyword is used, the variable name must be an atom-style or atomfile-style variable. The variable is evaluated and atoms whose per-atom values are 0.0, are removed from the dynamic group.

If the *property* keyword is used, the name refers to a custom integer or floating point per-atom vector defined via the *fix property/atom* command. This means the values in the vector can be read as part of a data file with the *read_data* command or specified with the *set* command. Or accessed and changed via the *library interface to LAMMPS*, or by styles you add to LAMMPS (pair, fix, compute, etc) which access the custom vector and modify its values. Which means the values can be modified between or during simulations. Atoms whose values in the custom vector are zero are removed from the dynamic group. Note that the name of the custom per-atom vector is specified just as *name*, not as *i_name* or *d_name* as it is for other commands that use different kinds of custom atom vectors or arrays as arguments.

The assignment of atoms to a dynamic group is done at the beginning of each run and on every timestep that is a multiple of *N*, which is the argument for the *every* keyword (*N* = 1 is the default). For an energy minimization, via the *minimize* command, an assignment is made at the beginning of the minimization, but not during the iterations of the minimizer.

The point in the timestep at which atoms are assigned to a dynamic group is after interatomic forces have been computed, but before any fixes which alter forces or otherwise update the system have been invoked. This means that atom positions have been updated, neighbor lists and ghost atoms are current, and both intermolecular and intramolecular forces have been calculated based on the new coordinates. Thus the region criterion, if applied, should be accurate. Also, any computes invoked by an atom-style variable should use updated information for that timestep (e.g., potential energy/atom or coordination number/atom). Similarly, fixes or computes which are invoked after that point in the timestep, should operate on the new group of atoms.

Note

If the *region* keyword is used to determine what atoms are in the dynamic group, atoms can move outside of the simulation box between reneighboring events. Thus if you want to include all atoms on the left side of the simulation box, you probably want to set the left boundary of the region to be outside the simulation box by some reasonable amount (e.g., up to the cutoff of the potential), else they may be excluded from the dynamic region.

Here is an example of using a dynamic group to shrink the set of atoms being integrated by using a spherical region with a variable radius (shrinking from 18 to 5 over the course of the run). This could be used to model a quench of the system, freezing atoms outside the shrinking sphere, then converting the remaining atoms to a static group and running further.

```
variable     nsteps equal 5000
variable     rad equal 18-(step/v_nsteps)*(18-5)
region       ss sphere 20 20 0 v_rad
group        mobile dynamic all region ss
fix          1 mobile nve
run          ${nsteps}
group        mobile static
run          ${nsteps}
```

Note

All fixes and computes take a group ID as an argument, but they do not all allow for use of a dynamic group. If you get an error message that this is not allowed, but feel that it should be for the fix or compute in question, then please post your reasoning to the LAMMPS forum at MatSci and we can look into changing it. The same applies if you come across inconsistent behavior when dynamic groups are allowed.

The *static* style removes the setting for a dynamic group, converting it to a static group (the default). The atoms in the static group are those currently in the dynamic group.

1.34.4 Restrictions

There can be no more than 32 groups defined at one time, including "all".

The parent group of a dynamic group cannot itself be a dynamic group.

1.34.5 Related commands

dump, *fix*, *region*, *velocity*

1.34.6 Default

All atoms belong to the "all" group.

1.35 group2ndx command

1.36 ndx2group command

1.36.1 Syntax

```
group2ndx file args
ndx2group file args
```

- file = name of index file to write out or read in
- args = zero or more group IDs may be appended

1.36.2 Examples

```
group2ndx allindex.ndx
group2ndx someindex.ndx upper lower mobile
ndx2group someindex.ndx
ndx2group someindex.ndx mobile
```

1.36.3 Description

Write or read a Gromacs style index file in text format that associates atom IDs with the corresponding group definitions. This index file can be used with in combination with Gromacs analysis tools or to import group definitions into the *fix colvars* input file.

It can also be used to save and restore group definitions for static groups using the individual atom IDs. This may be important if the original group definition depends on a region or otherwise on the geometry and thus cannot be easily recreated.

Another application would be to import atom groups defined for Gromacs simulation into LAMMPS. When translating Gromacs topology and geometry data to LAMMPS.

The *group2ndx* command will write group definitions to an index file. Without specifying any group IDs, all groups will be written to the index file. When specifying group IDs, only those groups will be written to the index file. In order to follow the Gromacs conventions, the group *all* will be renamed to *System* in the index file.

The *ndx2group* command will create of update group definitions from those stored in an index file. Without specifying any group IDs, all groups except *System* will be read from the index file and the corresponding groups recreated. If a group of the same name already exists, it will be completely reset. When specifying group IDs, those groups, if present, will be read from the index file and restored.

1.36.4 File Format

The file format is equivalent and compatible with what is produced by the Gromacs make_ndx command. and follows the Gromacs definition of an ndx file

Each group definition begins with the group name in square brackets with blanks, e.g. [water] and is then followed by the list of atom indices, which may be spread over multiple lines. Here is a small example file:

```
[ Oxygen ]
1  4  7
[ Hydrogen ]
2  3  5  6
8  9
[ Water ]
1 2 3 4 5 6 7 8 9
```

The index file defines 3 groups: Oxygen, Hydrogen, and Water and the latter happens to be the union of the first two.

1.36.5 Restrictions

These commands require that atoms have atom IDs, since this is the information that is written to the index file.

These commands are part of the EXTRA-COMMAND package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

1.36.6 Related commands

group, *dump*, *fix colvars*

1.36.7 Default

none

1.37 hyper command

1.37.1 Syntax

```
hyper N Nevent fix-ID compute-ID keyword values ...
```

- N = # of timesteps to run
- Nevent = check for events every this many steps
- fix-ID = ID of a fix that applies a global or local bias potential, can be NULL
- compute-ID = ID of a compute that identifies when an event has occurred
- zero or more keyword/value pairs may be appended
- keyword = *min* or *dump* or *rebond*

```
min values = etol ftol maxiter maxeval
  etol = stopping tolerance for energy, used in quenching
  ftol = stopping tolerance for force, used in quenching
  maxiter = max iterations of minimize, used in quenching
  maxeval = max number of force/energy evaluations, used in quenching
dump value = dump-ID
  dump-ID = ID of dump to trigger whenever an event takes place
rebond value = Nrebond
  Nrebond = frequency at which to reset bonds, even if no event has occurred
```

1.37.2 Examples

```
compute event all event/displace 1.0
fix HG mobile hyper/global 3.0 0.3 0.4 800.0
hyper 5000 100 HG event min 1.0e-6 1.0e-6 100 100 dump 1 dump 5
```

1.37.3 Description

Run a bond-boost hyperdynamics (HD) simulation where time is accelerated by application of a bias potential to one or more pairs of nearby atoms in the system. This command can be used to run both global and local hyperdynamics. In global HD a single bond within the system is biased on each timestep. In local HD multiple bonds (separated by a sufficient distance) can be biased simultaneously at each timestep. In the bond-boost hyperdynamics context, a "bond" is not a covalent bond between a pair of atoms in a molecule. Rather it is simply a pair of nearby atoms as discussed below.

Both global and local HD are described in *(Voter2013)* by Art Voter and collaborators. Similar to parallel replica dynamics (PRD), global and local HD are methods for performing accelerated dynamics that are suitable for infrequent-event systems that obey first-order kinetics. A good overview of accelerated dynamics methods (AMD) for such systems in given in *(Voter2002)* from the same group. To quote from the review paper: "The dynamical evolution is characterized by vibrational excursions within a potential basin, punctuated by occasional transitions between basins. The transition probability is characterized by p(t) = k*exp(-kt) where k is the rate constant."

Both HD and PRD produce a time-accurate trajectory that effectively extends the timescale over which a system can be simulated, but they do it differently. HD uses a single replica of the system and accelerates time by biasing the interaction potential in a manner such that each timestep is effectively longer. PRD creates Nr replicas of the system and runs dynamics on each independently with a normal unbiased potential until an event occurs in one of the replicas. The time between events is reduced by a factor of Nr replicas. For both methods, per CPU second, more physical time elapses and more events occur. See the *prd* page for more info about PRD.

An HD run has several stages, which are repeated each time an event occurs, as explained below. The logic for an HD run is as follows:

```
quench
create initial list of bonds

while (time remains):
  run dynamics for Nevent steps
  quench
  check for an event
  if event occurred: reset list of bonds
  restore pre-quench state
```

The list of bonds is the list of atom pairs of atoms that are within a short cutoff distance of each other after the system energy is minimized (quenched). This list is created and reset by a *fix hyper/global* or *fix hyper/local* command specified as *fix-ID*. At every dynamics timestep, the same fix selects one of more bonds to apply a bias potential to.

Note

The style of fix associated with the specified *fix-ID* determines whether you are running the global versus local hyperdynamics algorithm.

Dynamics (with the bias potential) is run continuously, stopping every *Nevent* steps to check if a transition event has occurred. The specified *N* for total steps must be a multiple of *Nevent*. check is performed by quenching the system and comparing the resulting atom coordinates to the coordinates from the previous basin.

A quench is an energy minimization and is performed by whichever algorithm has been defined by the *min_style* command. Minimization parameters may be set via the *min_modify* command and by the *min* keyword of the hyper command. The latter are the settings that would be used with the *minimize* command. Note that typically, you do not need to perform a highly-converged minimization to detect a transition event, though you may need to in order to prevent a set of atoms in the system from relaxing to a saddle point.

The event check is performed by a compute with the specified *compute-ID*. Currently there is only one compute that works with the hyper command, which is the *compute event/displace* command. Other event-checking computes may be added. *Compute event/displace* checks whether any atom in the compute group has moved further than a specified threshold distance. If so, an event has occurred.

If this happens, the list of bonds is reset, since some bond pairs are likely now too far apart, and new pairs are likely close enough to be considered a bond. The pre-quenched state of the system (coordinates and velocities) is restored, and dynamics continue.

At the end of the hyper run, a variety of statistics are output to the screen and logfile. These include info relevant to both global and local hyperdynamics, such as the number of events and the elapsed hyper time (accelerated time), And it includes info specific to one or the other, depending on which style of fix was specified by *fix-ID*.

The optional keywords operate as follows.

As explained above, the *min* keyword can be used to specify parameters for the quench. Their meaning is the same as for the *minimize* command

The *dump* keyword can be used to trigger a specific dump command with the specified *dump-ID* to output a snapshot each time an event is detected. It can be specified multiple times with different *dump-ID* values, as in the example above. These snapshots will be for the quenched state of the system on a timestep that is a multiple of *Nevent*, i.e. a timestep after the event has occurred. Note that any dump command in the input script will also output snapshots at whatever timestep interval it defines via its *N* argument; see the *dump* command for details. This means if you only want a particular dump to output snapshots when events are detected, you should specify its *N* as a value larger than the length of the hyperdynamics run.

As in the code logic above, the bond list is normally only reset when an event occurs. The *rebond* keyword will force a reset of the bond list every *Nrebond* steps, even if an event has not occurred. *Nrebond* must be a multiple of *Nevent*. This can be useful to check if more frequent resets alter event statistics, perhaps because the parameters chosen for defining what is a bond and what is an event are producing bad dynamics in the presence of the bias potential.

1.37.4 Restrictions

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

1.37.5 Related commands

fix hyper/global, *fix hyper/local*, *compute event/displace*, *prd*

1.37.6 Default

The option defaults are min = 0.1 0.1 40 50 and time = steps.

(Voter2013) S. Y. Kim, D. Perez, A. F. Voter, J Chem Phys, 139, 144110 (2013).

(Voter2002) Voter, Montalenti, Germann, Annual Review of Materials Research 32, 321 (2002).

1.38 if command

1.38.1 Syntax

```
if boolean then t1 t2 ... elif boolean f1 f2 ... elif boolean f1 f2 ... else e1 e2 ...
```

- boolean = a Boolean expression evaluated as TRUE or FALSE (see below)
- then = required word
- t1,t2,...,tN = one or more LAMMPS commands to execute if condition is met, each enclosed in quotes
- elif = optional word, can appear multiple times
- f1,f2,...,fN = one or more LAMMPS commands to execute if elif condition is met, each enclosed in quotes (optional arguments)
- else = optional argument
- e1,e2,...,eN = one or more LAMMPS commands to execute if no condition is met, each enclosed in quotes (optional arguments)

1.38.2 Examples

```
if "${steps} > 1000" then quit
if "${myString} == a10" then quit
if "$x <= $y" then "print 'X is smaller = $x'" else "print 'Y is smaller = $y'"
if "(${eng} > 0.0) || ($n < 1000)" then &
  "timestep 0.005" &
elif $n<10000 &
  "timestep 0.01" &
else &
  "timestep 0.02" &
  "print 'Max step reached'"
if "${eng} > ${eng_previous}" then "jump file1" else "jump file2"
```

1.38.3 Description

This command provides an if-then-else capability within an input script. A Boolean expression is evaluated and the result is TRUE or FALSE. Note that as in the examples above, the expression can contain variables, as defined by the *variable* command, which will be evaluated as part of the expression. Thus a user-defined formula that reflects the current state of the simulation can be used to issue one or more new commands.

If the result of the Boolean expression is TRUE, then one or more commands (t1, t2, ..., tN) are executed. If it is FALSE, then Boolean expressions associated with successive elif keywords are evaluated until one is found to be true, in which case its commands (f1, f2, ..., fN) are executed. If no Boolean expression is TRUE, then the commands associated with the else keyword, namely (e1, e2, ..., eN), are executed. The elif and else keywords and their associated commands are optional. If they are not specified and the initial Boolean expression is FALSE, then no commands are executed.

The syntax for Boolean expressions is described below.

Each command (t1, f1, e1, etc.) can be any valid LAMMPS input script command. If the command is more than one word, it must enclosed in quotes, so it will be treated as a single argument, as in the examples above.

Note

If a command itself requires a quoted argument (e.g., a *print* command), then double and single quotes can be used and nested in the usual manner, as in the examples above and below. The *Commands parse* page has more details on using quotes in arguments. Only one of level of nesting is allowed, but that should be sufficient for most use cases.

Note that by using the line continuation character "&", the if command can be spread across many lines, though it is still a single command:

```
if "$a < $b" then &
  "print 'Minimum value = $a'" &
  "run 1000" &
else &
  'print "Minimum value = $b"' &
  "minimize 0.001 0.001 1000 10000"
```

Note that if one of the commands to execute is *quit*, as in the first example above, then executing the command will cause LAMMPS to halt.

Note that by jumping to a label in the same input script, the if command can be used to break out of a loop. See the *variable delete* command for info on how to delete the associated loop variable, so that it can be re-used later in the input script.

Here is an example of a loop which checks every 1000 steps if the system temperature has reached a certain value, and if so, breaks out of the loop to finish the run. Note that any variable could be checked, so long as it is current on the timestep when the run completes. As explained on the *variable* doc page, this can be ensured by including the variable in thermodynamic output.

```
variable myTemp equal temp
label loop
variable a loop 1000
run 1000
if "${myTemp} < 300.0" then "jump SELF break"
next a
jump SELF loop
label break
print "ALL DONE"
```

Here is an example of a double loop which uses the if and *jump* commands to break out of the inner loop when a condition is met, then continues iterating through the outer loop.

```
label     loopa
variable  a loop 5
  label     loopb
  variable  b loop 5
    print     "A,B = $a,$b"
    run       10000
    if        "$b > 2" then "jump SELF break"
  next      b
  jump      in.script loopb
  label     break
  variable  b delete
next      a
```

(continues on next page)

(continued from previous page)

```
jump      SELF loopa
```

The Boolean expressions for the if and elif keywords have a C-like syntax. Note that each expression is a single argument within the if command. Thus if you want to include spaces in the expression for clarity, you must enclose the entire expression in quotes.

An expression is built out of numbers (which start with a digit or period or minus sign) or strings (which start with a letter and can contain alphanumeric characters, underscores, or forward slashes):

```
0.2, 100, 1.0e20, -15.4, ...
InP, myString, a123, ab_23_cd, lj/cut, ...
```

and Boolean operators:

A == B, A != B, A < B, A <= B, A > B, A >= B, A && B, A || B, A |^ B, !A

Each A and B is a number or string or a variable reference like $a or ${abc}, or A or B can be another Boolean expression.

Note that all variables used will be substituted for before the Boolean expression in evaluated. A variable can produce a number, like an *equal-style variable*, or it can produce a string, like an *index-style variable*.

The Boolean operators == and != can operate on a pair or strings or numbers. They cannot compare a number to a string. All the other Boolean operations can only operate on numbers.

Expressions are evaluated left to right and have the usual C-style precedence: the unary logical NOT operator ! has the highest precedence, the 4 relational operators <, <=, >, and >= are next; the two remaining relational operators == and != are next; then the logical AND operator &&; and finally the logical OR operator || and logical XOR (exclusive or) operator |^ have the lowest precedence. Parenthesis can be used to group one or more portions of an expression and/or enforce a different order of evaluation than what would occur with the default precedence.

When the six relational operators (first six in list above) compare two numbers, they return either a 1.0 or 0.0 depending on whether the relationship between A and B is TRUE or FALSE.

When the three logical operators (last three in list above) compare two numbers, they also return either a 1.0 or 0.0 depending on whether the relationship between A and B is TRUE or FALSE (or just A). The logical AND operator will return 1.0 if both its arguments are non-zero, else it returns 0.0. The logical OR operator will return 1.0 if either of its arguments is non-zero, else it returns 0.0. The logical XOR operator will return 1.0 if one of its arguments is zero and the other non-zero, else it returns 0.0. The logical NOT operator returns 1.0 if its argument is 0.0, else it returns 0.0. The 3 logical operators can only be used to operate on numbers, not on strings.

The overall Boolean expression produces a TRUE result if the numeric result is non-zero. If the result is zero, the expression result is FALSE.

Note

If the Boolean expression is a single numeric value with no Boolean operators, it will be FALSE if the value = 0.0, otherwise TRUE. If the Boolean expression is a single string, an error message will be issued.

1.38.4 Restrictions

none

1.38.5 Related commands

variable, *print*

1.38.6 Default

none

1.39 improper_coeff command

1.39.1 Syntax

```
improper_coeff N args
```

- N = numeric improper type (see asterisk form below), or type label
- args = coefficients for one or more improper types

1.39.2 Examples

```
improper_coeff 1 300.0 0.0
improper_coeff * 80.2 -1 2
improper_coeff *4 80.2 -1 2

labelmap improper 1 benzene
improper_coeff benzene 300.0 0.0
```

1.39.3 Description

Specify the improper force field coefficients for one or more improper types. The number and meaning of the coefficients depends on the improper style. Improper coefficients can also be set in the data file read by the *read_data* command or in a restart file.

N can be specified in one of two ways. An explicit numeric value can be used, as in the first example above. Or *N* can be a type label, which is an alphanumeric string defined by the *labelmap* command or in a section of a data file read by the *read_data* command.

For numeric values only, a wild-card asterisk can be used to set the coefficients for multiple improper types. This takes the form "*" or "*n" or "n*" or "m*n". If *N* = the number of improper types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

Note that using an improper_coeff command can override a previous setting for the same improper type. For example, these commands set the coeffs for all improper types, then overwrite the coeffs for just improper type 2:

```
improper_coeff * 300.0 0.0
improper_coeff 2 50.0 0.0
```

A line in a data file that specifies improper coefficients uses the exact same format as the arguments of the improper_coeff command in an input script, except that wild-card asterisks should not be used since coefficients for all *N* types must be listed in the file. For example, under the "Improper Coeffs" section of a data file, the line that corresponds to the first example above would be listed as

```
1 300.0 0.0
```

The *improper_style class2* is an exception to this rule, in that an additional argument is used in the input script to allow specification of the cross-term coefficients. See its doc page for details.

The list of all improper styles defined in LAMMPS is given on the *improper_style* doc page. They are also listed in more compact form on the *Commands improper* doc page.

On either of those pages, click on the style to display the formula it computes and its coefficients as specified by the associated improper_coeff command.

1.39.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

An improper style must be defined before any improper coefficients are set, either in the input script or in a data file.

1.39.5 Related commands

improper_style

1.39.6 Default

none

1.40 improper_style command

1.40.1 Syntax

```
improper_style style
```

- style = *none* or *hybrid* or *class2* or *cvff* or *harmonic*

1.40.2 Examples

```
improper_style harmonic
improper_style cvff
improper_style hybrid cvff harmonic
```

1.40.3 Description

Set the formula(s) LAMMPS uses to compute improper interactions between quadruplets of atoms, which remain in force for the duration of the simulation. The list of improper quadruplets is read in by a *read_data* or *read_restart* command from a data or restart file. Note that the ordering of the 4 atoms in an improper quadruplet determines the definition of the improper angle used in the formula for each style. See the doc pages of individual styles for details.

Hybrid models where impropers are computed using different improper potentials can be setup using the *hybrid* improper style.

The coefficients associated with an improper style can be specified in a data or restart file or via the *improper_coeff* command.

All improper potentials store their coefficient data in binary restart files which means improper_style and *improper_coeff* commands do not need to be re-specified in an input script that restarts a simulation. See the *read_restart* command for details on how to do this. The one exception is that improper_style *hybrid* only stores the list of sub-styles in the restart file; improper coefficients need to be re-specified.

Note

When both an improper and pair style is defined, the *special_bonds* command often needs to be used to turn off (or weight) the pairwise interaction that would otherwise exist between a group of 4 bonded atoms.

Here is an alphabetic list of improper styles defined in LAMMPS. Click on the style to display the formula it computes and coefficients specified by the associated *improper_coeff* command.

Click on the style to display the formula it computes, any additional arguments specified in the improper_style command, and coefficients specified by the associated *improper_coeff* command.

There are also additional accelerated pair styles included in the LAMMPS distribution for faster performance on CPUs, GPUs, and KNLs. The individual style names on the *Commands improper* page are followed by one or more of (g,i,k,o,t) to indicate which accelerated styles exist.

- *none* - turn off improper interactions
- *zero* - topology but no interactions
- *hybrid* - define multiple styles of improper interactions
- *amoeba* - AMOEBA out-of-plane improper
- *class2* - COMPASS (class 2) improper
- *cossq* - improper with a cosine squared term
- *cvff* - CVFF improper
- *distance* - improper based on distance between atom planes
- *distharm* - improper that is harmonic in the out-of-plane distance

- *fourier* - improper with multiple cosine terms
- *harmonic* - harmonic improper
- *inversion/harmonic* - harmonic improper with Wilson-Decius out-of-plane definition
- *ring* - improper which prevents planar conformations
- *umbrella* - DREIDING improper
- *sqdistharm* - improper that is harmonic in the square of the out-of-plane distance

---

1.40.4 Restrictions

Improper styles can only be set for atom_style choices that allow impropers to be defined.

Most improper styles are part of the MOLECULE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info. The doc pages for individual improper potentials tell if it is part of a package.

1.40.5 Related commands

improper_coeff

1.40.6 Default

```
improper_style none
```

1.41 include command

1.41.1 Syntax

```
include file
```

- file = filename of new input script to switch to

1.41.2 Examples

```
include newfile
include in.run2
```

1.41.3 Description

This command opens a new input script file and begins reading LAMMPS commands from that file. When the new file is finished, the original file is returned to. Include files can be nested as deeply as desired. If input script A includes script B, and B includes A, then LAMMPS could run for a long time.

If the filename is a variable (see the *variable* command), different processor partitions can run different input scripts.

1.41.4 Restrictions

none

1.41.5 Related commands

variable, *jump*

1.41.6 Default

none

1.42 info command

1.42.1 Syntax

```
info args
```

- args = one or more of the following keywords: *out*, *all*, *system*, *memory*, *communication*, *computes*, *dumps*, *fixes*, *groups*, *regions*, *variables*, *coeffs*, *styles*, *time*, *accelerator*, *fft* or *configuration*
- *out* values = *screen*, *log*, *append* filename, *overwrite* filename
- *styles* values = *all*, *angle*, *atom*, *bond*, *compute*, *command*, *dump*, *dihedral*, *fix*, *improper*, *integrate*, *kspace*, *minimize*, *pair*, *region*

1.42.2 Examples

```
info system
info groups computes variables
info all out log
info all out append info.txt
info styles all
info styles atom styles command
```

1.42.3 Description

Print out information about the current internal state of the running LAMMPS process. This can be helpful when debugging or validating complex input scripts. Several output categories are available and one or more output categories may be requested. All category keywords take no arguments, only *out* and *styles* take arguments as shown below. The keywords are cumulative, may be abbreviated, and unknown keywords are ignored.

The *out* flag controls where the output is sent. It can only be sent to one target. By default this is the screen, if it is active. The *log* argument selects the log file instead. With the *append* and *overwrite* option, followed by a filename, the output is written to that file, which is either appended to or overwritten, respectively.

The *all* flag activates printing all categories listed below.

The *configuration* category prints some information about the LAMMPS version as well as architecture and OS it is run on.

The *memory* category prints some information about the current memory allocation of MPI rank 0 (this the amount of dynamically allocated memory reported by LAMMPS classes). Where supported, also some OS specific information about the size of the reserved memory pool size (this is where malloc() and the new operator request memory from) and the maximum resident set size is reported (this is the maximum amount of physical memory occupied so far).

The *system* category prints a general system overview listing. This includes the unit style, atom style, number of atoms, bonds, angles, dihedrals, and impropers and the number of the respective types, box dimensions and properties, force computing styles and more.

The *communication* category prints a variety of information about communication and parallelization: the MPI library version level, the number of MPI ranks and OpenMP threads, the communication style and layout, the processor grid dimensions, ghost atom communication mode, cutoff, and related settings.

The *computes* category prints a list of all currently defined computes, their IDs and styles and groups they operate on.

The *dumps* category prints a list of all currently active dumps, their IDs, styles, filenames, groups, and dump frequencies.

The *fixes* category prints a list of all currently defined fixes, their IDs and styles and groups they operate on.

The *groups* category prints a list of all currently defined groups.

The *regions* category prints a list of all currently defined regions, their IDs and styles and whether "inside" or "outside" atoms are selected.

The *variables* category prints a list of all currently defined variables, their names, styles, definition and last computed value, if available.

The *coeffs* category prints a list for each defined force style (pair, bond, angle, dihedral, improper) indicating which of the corresponding coefficients have been set. This can be very helpful to debug error messages like "All pair coeffs are not set".

The *accelerator* category prints out information about compile time settings of included accelerator support for the GPU, KOKKOS, INTEL, and OPENMP packages.

Added in version 7Feb2024.

The *fft* category prints out information about the included 3d-FFT support. This lists the 3d-FFT engine, FFT precision, FFT library used by the FFT engine. If the KOKKOS package is included, the settings used for the KOKKOS package are displayed as well.

The *styles* category prints the list of styles available in the current LAMMPS binary. The *styles* keyword without option is the same as using the "all" option. One of the following options may be used to control which category of styles is printed out. To select multiple categories, the styles keyword needs to be used multiple times with the desired categories:

- all

- angle
- atom
- bond
- compute
- command
- dump
- dihedral
- fix
- improper
- integrate
- kspace
- minimize
- pair
- region

The *time* category prints the accumulated CPU and wall time for the process that writes output (usually MPI rank 0).

1.42.4 Restrictions

none

1.42.5 Related commands

print

1.42.6 Default

The *out* option has the default *screen*.

The *styles* option has the default *all*.

1.43 jump command

1.43.1 Syntax

```
jump file label
```

- file = filename of new input script to switch to
- label = optional label within file to jump to

1.43.2 Examples

```
jump newfile
jump in.run2 runloop
jump SELF runloop
```

1.43.3 Description

This command closes the current input script file, opens the file with the specified name, and begins reading LAMMPS commands from that file. Unlike the *include* command, the original file is not returned to, although by using multiple jump commands it is possible to chain from file to file or back to the original file.

If the word "SELF" is used for the filename, then the current input script is re-opened and read again.

Note

The SELF option is not guaranteed to work when the current input script is being read through stdin (standard input), e.g.

```
lmp_g++ < in.script
```

since the SELF option invokes the C-library rewind() call, which may not be supported for stdin on some systems or by some MPI implementations. This can be worked around by using the *-in command-line switch*, e.g.

```
lmp_g++ -in in.script
```

or by using the *-var command-line switch* to pass the script name as a variable to the input script. In the latter case, a *variable* called "fname" could be used in place of SELF, e.g.

```
lmp_g++ -var fname in.script < in.script
```

The second argument to the jump command is optional. If specified, it is treated as a label and the new file is scanned (without executing commands) until the label is found, and commands are executed from that point forward. This can be used to loop over a portion of the input script, as in this example. These commands perform 10 runs, each of 10000 steps, and create 10 dump files named file.1, file.2, etc. The *next* command is used to exit the loop after 10 iterations. When the "a" variable has been incremented for the tenth time, it will cause the next jump command to be skipped.

```
variable a loop 10
label loop
dump 1 all atom 100 file.$a
run 10000
undump 1
next a
jump in.lj loop
```

If the jump *file* argument is a variable, the jump command can be used to cause different processor partitions to run different input scripts. In this example, LAMMPS is run on 40 processors, with 4 partitions of 10 procs each. An in.file containing the example variable and jump command will cause each partition to run a different simulation.

```
mpirun -np 40 lmp_ibm -partition 4x10 -in in.file
```

(continues on next page)

(continued from previous page)

```
variable f world script.1 script.2 script.3 script.4
jump $f
```

Here is an example of a loop which checks every 1000 steps if the system temperature has reached a certain value, and if so, breaks out of the loop to finish the run. Note that any variable could be checked, so long as it is current on the timestep when the run completes. As explained on the *variable* doc page, this can be ensured by including the variable in thermodynamic output.

```
variable myTemp equal temp
label loop
variable a loop 1000
run 1000
if "${myTemp} < 300.0" then "jump SELF break"
next a
jump SELF loop
label break
print "ALL DONE"
```

Here is an example of a double loop which uses the if and *jump* commands to break out of the inner loop when a condition is met, then continues iterating through the outer loop.

```
label     loopa
variable  a loop 5
  label     loopb
  variable  b loop 5
    print     "A,B = $a,$b"
    run       10000
    if        "$b > 2" then "jump SELF break"
  next      b
  jump      in.script loopb
  label     break
  variable  b delete
next      a
jump      SELF loopa
```

1.43.4 Restrictions

If you jump to a file and it does not contain the specified label, LAMMPS will come to the end of the file and exit.

1.43.5 Related commands

variable, *include*, *label*, *next*

1.43.6 Default

none

1.44 kim command

1.44.1 Syntax

```
kim sub-command args
```

- sub-command = *init* or *interactions* or *query* or *param* or *property*
- args = arguments used by a particular sub-command

1.44.2 Examples

```
kim init args
kim interactions args
kim query args
kim param args
kim property args
```

1.44.3 Description

The *kim command* includes a set of sub-commands that allow LAMMPS users to use interatomic models (IM) (potentials and force fields) and their predictions for various physical properties archived in the Open Knowledgebase of Interatomic Models (OpenKIM) repository.

Using OpenKIM provides LAMMPS users with immediate access to a large number of verified IMs and their predictions. OpenKIM IMs have multiple benefits including reliability, reproducibility and convenience.

There are two types of IMs archived in OpenKIM:

1. The first type is called a *KIM Portable Model* (PM). A KIM PM is an independent computer implementation of an IM written in one of the languages supported by KIM (C, C++, Fortran) that conforms to the KIM Application Programming Interface (KIM API) Portable Model Interface (PMI) standard. A KIM PM will work seamlessly with any simulation code that supports the KIM API/PMI standard (including LAMMPS; see complete list of supported codes).
2. The second type is called a *KIM Simulator Model* (SM). A KIM SM is an IM that is implemented natively within a simulation code (*simulator*) that supports the KIM API Simulator Model Interface (SMI); in this case LAMMPS. A separate SM package is archived in OpenKIM for each parameterization of the IM, which includes all of the necessary parameter files, LAMMPS commands, and metadata (supported species, units, etc.) needed to run the IM in LAMMPS.

With these two IM types, OpenKIM can archive and test almost all IMs that can be used by LAMMPS. (It is easy to contribute new IMs to OpenKIM, see the upload instructions.)

OpenKIM IMs are uniquely identified by a KIM ID. The extended KIM ID consists of a human-readable prefix identifying the type of IM, authors, publication year, and supported species, separated by two underscores from the KIM ID itself, which begins with an IM code (*MO* for a KIM Portable Model, and *SM* for a KIM Simulator Model) followed by a unique 12-digit code and a 3-digit version identifier. By convention SM prefixes begin with *Sim_* to readily identify them.

```
SW_StillingerWeber_1985_Si__MO_405512056662_005
Sim_LAMMPS_ReaxFF_StrachanVanDuinChakraborty_2003_CHNO__SM_107643900657_001
```

Each OpenKIM IM has a dedicated "Model Page" on OpenKIM providing all the information on the IM including a title, description, authorship and citation information, test and verification check results, visualizations of results, a wiki with documentation and user comments, and access to raw files, and other information. The URL for the Model Page is constructed from the extended KIM ID of the IM:

```
https://openkim.org/id/extended_KIM_ID
```

For example, for the Stillinger-Weber potential listed above the Model Page is located at:

https://openkim.org/id/SW_StillingerWeber_1985_Si__MO_405512056662_005

See the current list of KIM PMs and SMs archived in OpenKIM. This list is sorted by species and can be filtered to display only IMs for certain species combinations.

See Obtaining KIM Models to learn how to install a pre-built binary of the OpenKIM Repository of Models.

Note

It is also possible to locally install IMs not archived in OpenKIM, in which case their names do not have to conform to the KIM ID format.

1.44.4 Using OpenKIM IMs with LAMMPS (*kim init*, *kim interactions*)

Two sub-commands are employed when using OpenKIM IMs in LAMMPS, one to select the IM and perform necessary initialization (*kim init*), and the second to set up the IM for use by executing any necessary LAMMPS commands (*kim interactions*). Both are required.

Syntax

```
kim init model user_units unitarg
kim interactions typeargs
```

- model = name of the KIM interatomic model (the KIM ID for models archived in OpenKIM)
- user_units = the LAMMPS *units* style assumed in the LAMMPS input script
- unitarg = *unit_conversion_mode* (optional)
- typeargs = atom type to species mapping (one entry per atom type) or *fixed_types* for models with a preset fixed mapping

Examples

```
kim init SW_StillingerWeber_1985_Si__MO_405512056662_005 metal
kim interactions Si

kim init Sim_LAMMPS_ReaxFF_StrachanVanDuinChakraborty_2003_CHNO__SM_107643900657_
↪001 real
kim init Sim_LAMMPS_ReaxFF_StrachanVanDuinChakraborty_2003_CHNO__SM_107643900657_
↪001 metal unit_conversion_mode
kim interactions C H O

kim init Sim_LAMMPS_IFF_PCFF_HeinzMishraLinEmami_2015Ver1v5_
↪FccmetalsMineralsSolventsPolymers__SM_039297821658_000 real
kim interactions fixed_types
```

See the *examples/kim* directory for example input scripts that use KIM PMs and KIM SMs.

OpenKIM IM Initialization (*kim init*)

The *kim* command followed by *init* sub-command must be issued **before** the simulation box is created (normally at the top of the file). This command sets the OpenKIM IM that will be used and may issue additional commands changing LAMMPS default settings that are required for using the selected IM (such as *units* or *atom_style*). If needed, those settings can be overridden, however, typically a script containing a *kim init* command would not include *units* and *atom_style* commands.

The required arguments of *kim init* are the *model* name of the IM to be used in the simulation (for an IM archived in OpenKIM this is its extended KIM ID, and the *user_units*, which are the LAMMPS *units style* used in the input script. (Any dimensioned numerical values in the input script and values read in from files are expected to be in the *user_units* system.)

The selected IM can be either a *KIM PM or a KIM SM*. For a KIM SM, the *kim init* command verifies that the SM is designed to work with LAMMPS (and not another simulation code). In addition, the LAMMPS version used for defining the SM and the LAMMPS version being currently run are printed to help diagnose any incompatible changes to input script or command syntax between the two LAMMPS versions.

Based on the selected model *kim init* may modify the *atom_style*. Some SMs have requirements for this setting. If this is the case, then *atom_style* will be set to the required style. Otherwise, the value is left unchanged (which in the absence of an *atom_style* command in the input script is the *default atom_style value*).

Regarding units, the *kim init* behaves in different ways depending on whether or not *unit conversion mode* is activated as indicated by the optional *unitarg* argument. If unit conversion mode is **not** active, then *user_units* must either match the required units of the IM or the IM must be able to adjust its units to match. (The latter is only possible with some KIM PMs; SMs can never adjust their units.) If a match is possible, the LAMMPS *units* command is called to set the units to *user_units*. If the match fails, the simulation is terminated with an error. The *kim init* command also sets the default value for the *skin* (extra distance beyond force cutoff) as 2.0 Angstroms and sets the default value for the *timestep* size as 1.0 femtosecond.

Here is an example of a LAMMPS script to compute the cohesive energy of a face-centered cubic (fcc) lattice for the MEAM potential by Pascuet and Fernandez (2015) for Al.

```
kim          init Sim_LAMMPS_MEAM_PascuetFernandez_2015_Al__SM_811588957187_000 metal
boundary     p p p
lattice      fcc 4.049
region       simbox block 0 1 0 1 0 1 units lattice
create_box   1 simbox
```

(continues on next page)

(continued from previous page)

```
create_atoms 1 box
mass         1 26.981539
kim          interactions Al
run          0
variable     Ec equal (pe/count(all))
print        "Cohesive Energy = ${Ec} eV"
```

The above script will end with an error in the *kim init* line if the IM is changed to another potential for Al that does not work with *metal* units. To address this, *kim init* offers the *unit_conversion_mode* as shown below.

If unit conversion mode *is* active, then *kim init* calls the LAMMPS *units* command to set the units to the IM's required or preferred units. Conversion factors between the IM's units and the *user_units* are defined for all *physical quantities* (mass, distance, etc.). (Note that converting to or from the "lj" unit style is not supported.) These factors are stored as *internal style variables* with the following standard names:

```
_u_mass
_u_distance
_u_time
_u_energy
_u_velocity
_u_force
_u_torque
_u_temperature
_u_pressure
_u_viscosity
_u_charge
_u_dipole
_u_efield
_u_density
```

If desired, the input script can be designed to work with these conversion factors so that the script will work without change with any OpenKIM IM. (This approach is used in the OpenKIM Testing Framework.)

For example, the script given above for the cohesive energy of fcc Al can be rewritten to work with any IM regardless of units. The following script constructs an fcc lattice with a lattice parameter defined in meters, computes the total energy, and prints the cohesive energy in Joules regardless of the units of the IM.

```
kim          init Sim_LAMMPS_MEAM_PascuetFernandez_2015_Al__SM_811588957187_000 si unit_
↪conversion_mode
boundary     p p p
lattice      fcc $(4.049e-10*v__u_distance)
region       simbox block 0 1 0 1 0 1 units lattice
create_box   1 simbox
create_atoms 1 box
mass         1 $(4.480134e-26*v__u_mass)
kim          interactions Al
neighbor     $(2e-10*v__u_distance) bin
run          0
variable     Ec_in_J equal (pe/count(all))/v__u_energy
print        "Cohesive Energy = ${Ec_in_J} J"
```

Note the multiplication by *v__u_distance* and *v__u_mass* to convert from SI units (specified in the *kim init* command) to whatever units the IM uses (metal in this case), and the division by *v__u_energy* to convert from the IM's energy units to SI units (Joule). This script will work correctly for any IM for Al (KIM PM or SM) selected by the *kim init*

command.

Care must be taken to apply unit conversion to dimensional variables read in from a file. For example, if a configuration of atoms is read in from a dump file using the *read_dump* command, the following can be done to convert the box and all atomic positions to the correct units:

```
change_box all x scale ${_u_distance} &
           y scale ${_u_distance} &
           z scale ${_u_distance} &
           xy final $(xy*v__u_distance) &
           xz final $(xz*v__u_distance) &
           yz final $(yz*v__u_distance) &
           remap
```

Note

Unit conversion will only work if the conversion factors are placed in all appropriate places in the input script. It is up to the user to do this correctly.

OpenKIM IM Execution (*kim interactions*)

The second and final step in using an OpenKIM IM is to execute the *kim interactions* command. This command must be preceded by a *kim init* command and a command that defines the number of atom types *N* (such as *create_box*). The *kim interactions* command has one argument *typeargs*. This argument contains either a list of *N* chemical species, which defines a mapping between atom types in LAMMPS to the available species in the OpenKIM IM, or the keyword *fixed_types* for models that have a preset fixed mapping (i.e. the mapping between LAMMPS atom types and chemical species is defined by the model and cannot be changed). In the latter case, the user must consult the model documentation to see how many atom types there are and how they map to the chemical species.

For example, consider an OpenKIM IM that supports Si and C species. If the LAMMPS simulation has four atom types, where the first three are Si, and the fourth is C, the following *kim interactions* command would be used:

```
kim interactions Si Si Si C
```

Alternatively, for a model with a fixed mapping the command would be:

```
kim interactions fixed_types
```

The *kim interactions* command performs all the necessary steps to set up the OpenKIM IM selected in the *kim init* command. The specific actions depend on whether the IM is a KIM PM or a KIM SM. For a KIM PM, a *pair_style kim* command is executed followed by the appropriate *pair_coeff* command. For example, for the Ercolessi and Adams (1994) KIM PM for Al set by the following commands:

```
kim init EAM_Dynamo_ErcolessiAdams_1994_Al__MO_123629422045_005 metal
...
...  box specification lines skipped
...
kim interactions Al
```

the *kim interactions* command executes the following LAMMPS input commands:

```
pair_style kim EAM_Dynamo_ErcolessiAdams_1994_Al__MO_123629422045_005
pair_coeff * * Al
```

For a KIM SM, the generated input commands may be more complex and require that LAMMPS is built with the required packages included for the type of potential being used. The set of commands to be executed is defined in the SM specification file, which is part of the SM package. For example, for the Strachan et al. (2003) ReaxFF SM set by the following commands:

```
kim init Sim_LAMMPS_ReaxFF_StrachanVanDuinChakraborty_2003_CHNO__SM_107643900657_
↪000 real
...
...  box specification lines skipped
...
kim interactions C H N O
```

the *kim interactions* command executes the following LAMMPS input commands:

```
pair_style reaxff lmp_control safezone 2.0 mincap 100
pair_coeff * * ffield.reax.rdx C H N O
fix reaxqeq all qeq/reaxff 1 0.0 10.0 1.0e-6 param.qeq
```

Note

The files *lmp_control*, *ffield.reax.rdx* and *param.qeq* are specific to the Strachan et al. (2003) ReaxFF parameterization and are archived as part of the SM package in OpenKIM.

Note

Parameters like cutoff radii and charge tolerances, which have an effect on IM predictions, are also included in the SM definition ensuring reproducibility.

Note

When using *kim init* and *kim interactions* to select and set up an OpenKIM IM, other LAMMPS commands for the same functions (such as pair_style, pair_coeff, bond_style, bond_coeff, fixes related to charge equilibration, etc.) should normally not appear in the input script.

Note

kim interactions must be called each time after the *change_box* command to provide the correct settings (it should be called with the same *typeargs* as the first call.) The reason is that changing a periodic boundary to a non-periodic one, or in general, using the *change_box* command after the interactions are set via *kim interactions* or *pair_coeff* commands might affect some of the settings. For example, SM models containing Coulombic terms in the interactions require different settings if a periodic boundary changes to a non-periodic one. In other cases, the second call to *kim interactions* does not affect any other settings.

1.44.5 Using OpenKIM Web Queries in LAMMPS (*kim query*)

The *kim query* command performs a web query to retrieve the predictions of an IM set by *kim init* for material properties archived in OpenKIM.

Syntax

```
kim query variable formatarg query_function queryargs
```

- variable(s) = single name or list of names of (string style) LAMMPS variable(s) where a query result or parameter get result is stored. Variables that do not exist will be created by the command
- formatarg = *list* or *split* or *index* (optional)

  list = returns a single string with a list of space separated values
  (e.g. "1.0 2.0 3.0"), which is placed in a LAMMPS variable as
  defined by the variable argument. [default]
  split = returns the values separately in new variables with names based
  on the prefix specified in variable and a number appended to
  indicate which element in the list of values is in the variable
  index = returns a variable style index that can be incremented via the
  next command. This enables the construction of simple loops

- query_function = name of the OpenKIM web API query function to be used
- queryargs = a series of *keyword=value* pairs that represent the web query; supported keywords depend on the query function

Examples

```
kim query a0 get_lattice_constant_cubic crystal=[fcc] species=[Al] units=[angstrom]
kim query model index get_available_models species=[Al] potential_type=[eam]
```

The result of the query is stored in one or more *string style variables* as determined by the optional *formatarg* argument. For the "list" setting of *formatarg* (or if *formatarg* is not specified), the result is returned as a space-separated list of values in *variable*. The *formatarg* keyword "split" separates the result values into individual variables of the form *prefix_I*, where *prefix* is set to the *kim query variable* argument and *I* ranges from 1 to the number of returned values. The number and order of the returned values is determined by the type of query performed. The *formatarg* keyword "index" returns a *variable style index* that can be incremented via the *next* command. This enables the construction of simple loops over the returned values by the type of query performed.

Note

kim query only supports queries that return a single result or an array of values. More complex queries that return a JSON structure are not currently supported. An attempt to use *kim query* in such cases will generate an error.

The second required argument *query_function* is the name of the query function to be called (e.g. *get_lattice_constant_cubic*). All following *arguments* are parameters handed over to the web query in the format *keyword=value*, where *value* is always an array of one or more comma-separated items in brackets. The list of supported keywords and the type and format of their values depend on the query function used. The current list of query functions is available on the OpenKIM webpage at https://openkim.org/doc/usage/kim-query.

Note

All query functions, except *get_available_models*, require the *model* keyword, which identifies the IM whose predictions are being queried. *kim query* automatically generates the *model* keyword based on the IM set in by *kim init*, and it can be overwritten if specified as an argument to the *kim query*. Where *kim init* is not specified, the *model* keyword must be provided as an argument to the *kim query*.

Note

Each *query_function* is associated with a default method (implemented as a KIM Test) used to compute this property. In cases where there are multiple methods in OpenKIM for computing a property, a *method* keyword can be provided to select the method of choice. See the query documentation to see which methods are available for a given *query_function*.

kim query Usage Examples and Further Clarifications

The data obtained by *kim query* commands can be used as part of the setup or analysis phases of LAMMPS simulations. Some examples are given below.

Define an equilibrium fcc crystal

```
kim      init EAM_Dynamo_ErcolessiAdams_1994_Al__MO_123629422045_005 metal
boundary p p p
kim      query a0 get_lattice_constant_cubic crystal=[fcc] species=[Al] units=[angstrom]
lattice  fcc ${a0}
...
```

```
units   metal
kim     query a0 get_lattice_constant_cubic crystal=[fcc] species=[Al] units=[angstrom] model=[EAM_
↪Dynamo_ErcolessiAdams_1994_Al__MO_123629422045_005]
lattice fcc ${a0}
...
```

The *kim query* command retrieves from OpenKIM the equilibrium lattice constant predicted by the Ercolessi and Adams (1994) potential for the fcc structure and places it in variable *a0*. This variable is then used on the next line to set up the crystal. By using *kim query*, the user is saved the trouble and possible error of tracking this value down, or of having to perform an energy minimization to find the equilibrium lattice constant.

Note

In *unit_conversion_mode* the results obtained from a *kim query* would need to be converted to the appropriate units system. For example, in the above script, the lattice command would need to be changed to: "lattice fcc $(v_a0*v__u_distance)".

Define an equilibrium hcp crystal

```
kim      init EAM_Dynamo_MendelevAckland_2007v3_Zr__MO_004835508849_000 metal
boundary p p p
kim      query latconst split get_lattice_constant_hexagonal crystal=[hcp] species=[Zr] units=[angstrom]
lattice  custom ${latconst_1} a1 0.5 -0.866025 0 a2 0.5 0.866025 0 a3 0 0 $(latconst_2/latconst_1) &
```

(continues on next page)

(continued from previous page)

```
        basis 0.333333 0.666666 0.25 basis 0.666666 0.333333 0.75
...
```

In this case the *kim query* returns two arguments (since the hexagonal close packed (hcp) structure has two independent lattice constants). The *formatarg* keyword "split" places the two values into the variables *latconst_1* and *latconst_2*. (These variables are created if they do not already exist.)

Define a crystal at finite temperature accounting for thermal expansion

```
kim      init EAM_Dynamo_ErcolessiAdams_1994_Al__MO_123629422045_005 metal
boundary p p p
kim      query a0 get_lattice_constant_cubic crystal=[fcc] species=[Al] units=[angstrom]
kim      query alpha get_linear_thermal_expansion_coefficient_cubic crystal=[fcc] species=[Al]␣
↪units=[1/K] temperature=[293.15] temperature_units=[K]
variable DeltaT equal 300
lattice  fcc $(v_a0*v_alpha*v_DeltaT)
...
```

As in the previous example, the equilibrium lattice constant is obtained for the Ercolessi and Adams (1994) potential. However, in this case the crystal is scaled to the appropriate lattice constant at room temperature (293.15 K) by using the linear thermal expansion constant predicted by the potential.

Note

When passing numerical values as arguments (as in the case of the temperature in the above example) it is also possible to pass a tolerance indicating how close to the value is considered a match. If no tolerance is passed a default value is used. If multiple results are returned (indicating that the tolerance is too large), *kim query* will return an error. See the query documentation to see which numerical arguments and tolerances are available for a given *query_function*.

Compute defect formation energy

```
kim      init EAM_Dynamo_ErcolessiAdams_1994_Al__MO_123629422045_005 metal
...
... Build fcc crystal containing some defect and compute the total energy
... which is stored in the variable *Etot*
...
kim      query Ec get_cohesive_energy_cubic crystal=[fcc] species=[Al] units=[eV]
variable Eform equal ${Etot} - count(all)*${Ec}
...
```

The defect formation energy *Eform* is computed by subtracting the ideal fcc cohesive energy of the atoms in the system from *Etot*. The ideal fcc cohesive energy of the atoms is obtained from OpenKIM for the Ercolessi and Adams (1994) potential.

Retrieve equilibrium fcc crystal of all EAM potentials that support a specific species

```
kim   query model index get_available_models species=[Al] potential_type=[eam]
label model_loop
kim   query latconst get_lattice_constant_cubic crystal=[fcc] species=[Al] units=[angstrom] model=[$
↪{model}]
print "FCC lattice constant (${model} potential) = ${latconst}"
...
```

(continues on next page)

(continued from previous page)

```
... do something with current value of latconst
...
next model
jump SELF model_loop
```

In this example, the *index* mode of *formatarg* is used. The first *kim query* returns the list of all available EAM potentials that support the *Al* species and archived in OpenKIM. The result of the query operation is stored in the LAMMPS variable *model* as an index *variable*. This variable is used later to access the values one at a time within a loop as shown in the example. The second *kim query* command retrieves from OpenKIM the equilibrium lattice constant predicted by each potential for the fcc structure and places it in variable *latconst*.

Note

kim query commands return results archived in OpenKIM. These results are obtained using programs for computing material properties (KIM Tests and KIM Test Drivers) that were contributed to OpenKIM. In order to give credit to Test developers, the number of times results from these programs are queried is tracked. No other information about the nature of the query or its source is recorded.

1.44.6 Accessing KIM Model Parameters from LAMMPS (*kim param*)

All IMs are functional forms containing a set of parameters. These parameters' values are typically selected to best reproduce a training set of quantum mechanical calculations or available experimental data. For example, a Lennard-Jones potential intended to model argon might have the values of its two parameters, epsilon, and sigma, fit to the dimer dissociation energy or thermodynamic properties at a critical point of the phase diagram.

Normally a user employing an IM should not modify its parameters since, as noted above, these are selected to reproduce material properties. However, there are cases where accessing and modifying IM parameters is desired, such as for assessing uncertainty, fitting an IM, or working with an ensemble of IMs. As explained *above*, IMs archived in OpenKIM are either Portable Models (PMs) or Simulator Models (SMs). KIM PMs are complete independent implementations of an IM, whereas KIM SMs are wrappers to an IM implemented within LAMMPS. Two different mechanisms are provided for accessing IM parameters in these two cases:

- For a KIM PM, the *kim param* command can be used to *get* and *set* the values of the PM's parameters as explained below.
- For a KIM SM, the user should consult the documentation page for the specific IM and follow instructions there for how to modify its parameters (if possible).

The *kim param get* and *kim param set* commands provide an interface to access and change the parameters of a KIM PM that "publishes" its parameters and makes them publicly available (see the KIM API documentation for details).

Note

The *kim param set/get* command must be preceded by a *kim interactions* command (or alternatively by a *pair_style kim* and *pair_coeff* commands). The *kim param set* command may be used wherever a *pair_coeff* command may occur.

Syntax

```
kim param get param_name index_range variable formatarg
kim param set param_name index_range values
```

- param_name = name of a KIM portable model parameter (which is published by the PM and available for access). The specific string used to identify a parameter is defined by the PM. For example, for the Stillinger-Weber (SW) potential in OpenKIM, the parameter names are *A, B, p, q, sigma, gamma, cutoff, lambda, costheta0*
- index_range = KIM portable model parameter index range (an integer for a single element, or pair of integers separated by a colon for a range of elements)
- variable(s) = single name or list of names of (string style) LAMMPS variable(s) where a query result or parameter get result is stored. Variables that do not exist will be created by the command
- formatarg = *list* or *split* or *explicit* (optional)

  list = returns a single string with a list of space separated values
  (e.g. "1.0 2.0 3.0"), which is placed in a LAMMPS variable as
  defined by the variable argument
  split = returns the values separately in new variables with names based
  on the prefix specified in variable and a number appended to
  indicate which element in the list of values is in the variable
  explicit = returns the values separately in one more more variable names
  provided as arguments that precede formatarg (default)
- values = new value(s) to replace the current value(s) of a KIM portable model parameter

Note

The list of all the parameters that a PM exposes for access/mutation are automatically written to the lammps log file when *kim init* is called.

Each published parameter of a KIM PM takes the form of an array of numerical values. The array can contain one element for a single-valued parameter, or a set of values. For example, the multispecies SW potential for the Zn-Cd-Hg-S-Se-Te system has the same parameter names as the single-species SW potential, but each parameter array contains 21 entries that correspond to the parameter values used for each pairwise combination of the model's six supported species (this model does not have parameters specific to individual ternary combinations of its supported species).

The *index_range* argument may either be an integer referring to a specific element within the array associated with the parameter specified by *param_name*, or a pair of integers separated by a colon that refer to a slice of this array. In both cases, one-based indexing is used to refer to the entries of the array.

The result of a *get* operation for a specific *index_range* is stored in one or more *LAMMPS string style variables* as determined by the optional *formatarg* argument *documented above.* If not specified, the default for *formatarg* is "explicit" for the *kim param* command.

For the case where the result is an array with multiple values (i.e. *index_range* contains a range), the optional "split" or "explicit" *formatarg* keywords can be used to separate the results into multiple variables; see the examples below. Multiple parameters can be retrieved with a single call to *kim param get* by repeating the argument list following *get*.

For a *set* operation, the *values* argument contains the new value(s) for the element(s) of the parameter specified by *index_range*. For the case where multiple values are being set, *values* contains a set of values separated by spaces. Multiple parameters can be set with a single call to *kim param set* by repeating the argument list following *set*.

kim param Usage Examples and Further Clarifications

Examples of getting and setting KIM PM parameters with further clarifications are provided below.

Getting a scalar parameter

```
kim init SW_StillingerWeber_1985_Si__MO_405512056662_005 metal
...
kim interactions Si
kim param get A 1 VARA
```

or

```
...
pair_style kim SW_StillingerWeber_1985_Si__MO_405512056662_005
pair_coeff * * Si
kim param get A 1 VARA
```

In these cases, the value of the SW *A* parameter is retrieved and placed in the LAMMPS variable *VARA*. The variable *VARA* can be used in the remainder of the input script in the same manner as any other LAMMPS variable.

Getting multiple scalar parameters with a single call

```
...
kim interactions Si
kim param get A 1 VARA B 1 VARB
```

In this example, it is shown how to retrieve the *A* and *B* parameters of the SW potential and store them in the LAMMPS variables *VARA* and *VARB*.

Getting a range of values from a parameter

There are several options when getting a range of values from a parameter determined by the *formatarg* argument.

```
kim init SW_ZhouWardMartin_2013_CdTeZnSeHgS__MO_503261197030_002 metal
...
kim interactions Te Zn Se
kim param get lambda 7:9 LAM_TeTe LAM_TeZn LAM_TeSe
```

In this case, *formatarg* is not specified and therefore the default "explicit" mode is used. (The behavior would be the same if the word *explicit* were added after *LAM_TeSe*.) Elements 7, 8 and 9 of parameter lambda retrieved by the *get* operation are placed in the LAMMPS variables *LAM_TeTe*, *LAM_TeZn* and *LAM_TeSe*, respectively.

Note

In the above example, elements 7-9 of the lambda parameter correspond to Te-Te, Te-Zm and Te-Se interactions. This can be determined by visiting the model page for the specified potential and looking at its parameter file linked to at the bottom of the page (file with .param ending) and consulting the README documentation provided with the driver for the PM being used. A link to the driver is provided at the top of the model page.

```
...
kim       interactions Te Zn Se
kim       param get lambda 15:17 LAMS list
variable LAM_VALUE index ${LAMS}
label    loop_on_lambda
```

(continues on next page)

(continued from previous page)

```
...
...     do something with the current value of lambda
...
next    LAM_VALUE
jump    SELF loop_on_lambda
```

In this case, the "list" mode of *formatarg* is used. The result of the *get* operation is stored in the LAMMPS variable *LAMS* as a string containing the three retrieved values separated by spaces, e.g "1.0 2.0 3.0". This can be used in LAMMPS with an *index* variable to access the values one at a time within a loop as shown in the example. At each iteration of the loop *LAM_VALUE* contains the current value of lambda.

```
...
kim interactions Te Zn Se
kim param get lambda 15:17 LAM split
```

In this case, the "split" mode of *formatarg* is used. The three values retrieved by the *get* operation are stored in the three LAMMPS variables *LAM_15*, *LAM_16* and *LAM_17*. The provided name "LAM" is used as prefix and the location in the lambda array is appended to create the variable names.

Setting a scalar parameter

```
kim init SW_StillingerWeber_1985_Si__MO_405512056662_005 metal
...
kim interactions Si
kim param set gamma 1 2.6
```

Here, the SW potential's gamma parameter is set to 2.6. Note that the *get* and *set* commands work together, so that a *get* following a *set* operation will return the new value that was set. For example,

```
...
kim interactions Si
kim param get gamma 1 ORIG_GAMMA
kim param set gamma 1 2.6
kim param get gamma 1 NEW_GAMMA
...
print "original gamma = ${ORIG_GAMMA}, new gamma = ${NEW_GAMMA}"
```

Here, *ORIG_GAMMA* will contain the original gamma value for the SW potential, while *NEW_GAMMA* will contain the value 2.6.

Setting multiple scalar parameters with a single call

```
kim      init SW_ZhouWardMartin_2013_CdTeZnSeHgS__MO_503261197030_002 metal
...
kim      interactions Cd Te
variable VARG equal 2.6
variable VARS equal 2.0951
kim      param set gamma 1 ${VARG} sigma 3 ${VARS}
```

In this case, the first element of the *gamma* parameter and third element of the *sigma* parameter are set to 2.6 and 2.0951, respectively. This example also shows how LAMMPS variables can be used when setting parameters.

Setting a range of values of a parameter

```
kim init SW_ZhouWardMartin_2013_CdTeZnSeHgS__MO_503261197030_002 metal
...
kim interactions Cd Te Zn Se Hg S
kim param set sigma 2:6 2.35214 2.23869 2.04516 2.43269 1.80415
```

In this case, elements 2 through 6 of the parameter *sigma* are set to the values 2.35214, 2.23869, 2.04516, 2.43269 and 1.80415 in order.

1.44.7 Writing material properties in standard KIM Property Instance format (*kim property*)

The OpenKIM system includes a collection of Tests (material property calculation codes), Models (interatomic potentials), Predictions, and Reference Data (DFT or experiments). Specifically, a KIM Test is a computation that when coupled with a KIM Model generates the prediction of that model for a specific material property rigorously defined by a KIM Property Definition (see the KIM Properties Framework for further details). A prediction of a material property for a given model is a specific numerical realization of a property definition, referred to as a "Property Instance." The objective of the *kim property* command is to make it easy to output material properties in a standardized, machine readable, format that can be easily ingested by other programs. Additionally, it aims to make it as easy as possible to convert a LAMMPS script that computes a material property into a KIM Test that can then be uploaded to openkim.org

A developer interested in creating a KIM Test using a LAMMPS script should first determine whether a property definition that applies to their calculation already exists in OpenKIM by searching the properties page. If none exists, it is possible to use a locally defined property definition contained in a file until it can be uploaded to the official repository (see below). Once one or more applicable property definitions have been identified, the *kim property create*, *kim property modify*, *kim property remove*, and *kim property destroy*, commands provide an interface to create, set, modify, remove, and destroy instances of them within a LAMMPS script.

Syntax

```
kim property create  instance_id property_id
kim property modify  instance_id key key_name key_name_key key_name_value
kim property remove  instance_id key key_name
kim property destroy instance_id
kim property dump    file
```

- instance_id = a positive integer identifying the KIM property instance; (note that the results file can contain multiple property instances)
- property_id = identifier of a KIM Property Definition, which can be (1) a property short name, (2) the full unique ID of the property (including the contributor and date), (3) a file name corresponding to a local property definition file
- key_name = one of the keys belonging to the specified KIM property definition
- key_name_key = a key belonging to a key-value pair (standardized in the KIM Properties Framework)
- key_name_value = value to be associated with a key_name_key in a key-value pair
- file = name of a file to write the currently defined set of KIM property instances to

Examples of each of the three *property_id* cases are shown below,

```
kim property create 1 atomic-mass
kim property create 2 cohesive-energy-relation-cubic-crystal
```

```
kim property create 1 tag:brunnels@noreply.openkim.org,2016-05-11:property/atomic-mass
kim property create 2 tag:staff@noreply.openkim.org,2014-04-15:property/cohesive-energy-relation-cubic-
↪crystal
```

```
kim property create 1 new-property.edn
kim property create 2 /home/mary/marys-kim-properties/dissociation-energy.edn
```

In the last example, "new-property.edn" and "/home/mary/marys-kim-properties/dissociation-energy.edn" are the names of files that contain user-defined (local) property definitions.

A KIM property instance takes the form of a "map", i.e. a set of key-value pairs akin to Perl's hash, Python's dictionary, or Java's Hashtable. It consists of a set of property key names, each of which is referred to here by the *key_name* argument, that are defined as part of the relevant KIM Property Definition and include only lowercase alphanumeric characters and dashes. The value paired with each property key is itself a map whose possible keys are defined as part of the KIM Properties Framework; these keys are referred to by the *key_name_key* argument and their associated values by the *key_name_value* argument. These values may either be scalars or arrays, as stipulated in the property definition.

Note

Each map assigned to a *key_name* must contain the *key_name_key* "source-value" and an associated *key_name_value* of the appropriate type (as defined in the relevant KIM Property Definition). For keys that are defined as having physical units, the "source-unit" *key_name_key* must also be given a string value recognized by GNU units.

Once a *kim property create* command has been given to instantiate a property instance, maps associated with the property's keys can be edited using the *kim property modify* command. In using this command, the special keyword "key" should be given, followed by the property key name and the key-value pair in the map associated with the key that is to be set. For example, the atomic-mass property definition consists of two property keys named "mass" and "species." An instance of this property could be created like so:

```
kim property create 1 atomic-mass
kim property modify 1 key species source-value Al
kim property modify 1 key mass    source-value 26.98154
kim property modify 1 key mass    source-unit amu
```

or, equivalently,

```
kim property create 1 atomic-mass
kim property modify 1 key species source-value Al       &
                      key mass    source-value 26.98154 &
                                  source-unit  amu
```

kim property Usage Examples and Further Clarifications

Create

```
kim property create instance_id property_id
```

The *kim property create* command takes as input a property instance ID and the property definition name, and creates an initial empty property instance data structure. For example,

```
kim property create 1 atomic-mass
kim property create 2 cohesive-energy-relation-cubic-crystal
```

creates an empty property instance of the "atomic-mass" property definition with instance ID 1 and an empty instance of the "cohesive-energy-relation-cubic-crystal" property with ID 2. A list of published property definitions in OpenKIM can be found on the properties page.

One can also provide the name of a file in the current working directory or the path of a file containing a valid property definition. For example,

```
kim property create 1 new-property.edn
```

where "new-property.edn" refers to a file name containing a new property definition that does not exist in OpenKIM.

If the *property_id* given cannot be found in OpenKIM and no file of this name containing a valid property definition can be found, this command will produce an error with an appropriate message. Calling *kim property create* with the same instance ID multiple times will also produce an error.

Modify

```
kim property modify instance_id key key_name key_name_key key_name_value
```

The *kim property modify* command incrementally builds the property instance by receiving property definition keys along with associated arguments. Each *key_name* is associated with a map containing one or more key-value pairs (in the form of *key_name_key-key_name_value* pairs). For example,

```
kim property modify 1 key species source-value Al
kim property modify 1 key mass    source-value 26.98154
kim property modify 1 key mass    source-unit  amu
```

where the special keyword "key" is followed by a *key_name* ("species" or "mass" in the above) and one or more key-value pairs. These key-value pairs may continue until either another "key" keyword is given or the end of the command line is reached. Thus, the above could equivalently be written as

```
kim property modify 1 key species source-value Al      &
                      key mass    source-value 26.98154 &
                      key mass    source-unit  amu
```

As an example of modifying multiple key-value pairs belonging to the map of a single property key, the following command modifies the map of the "cohesive-potential-energy" property key to contain the key "source-unit" which is assigned a value of "eV" and the key "digits" which is assigned a value of 5,

```
kim property modify 2 key cohesive-potential-energy source-unit eV digits 5
```

Note

The relevant data types of the values in the map are handled automatically based on the specification of the key in the KIM Property Definition. In the example above, this means that the value "eV" will automatically be interpreted as a string while the value 5 will be interpreted as an integer.

The values contained in maps can either be scalars, as in all of the examples above, or arrays depending on which is stipulated in the corresponding Property Definition. For one-dimensional arrays, a single one-based index must be supplied that indicates which element of the array is to be modified. For multidimensional arrays, multiple indices must be given depending on the dimensionality of the array.

Note

All array indexing used by *kim property modify* is one-based, i.e. the indices are enumerated 1, 2, 3, ...

Note

The dimensionality of arrays are defined in the the corresponding Property Definition. The extent of each dimension of an array can either be a specific finite number or indefinite and determined at run time. If an array has a fixed extent, attempting to modify an out-of-range index will fail with an error message.

For example, the "species" property key of the cohesive-energy-relation-cubic-crystal property is a one-dimensional array that can contain any number of entries based on the number of atoms in the unit cell of a given cubic crystal. To assign an array containing the string "Al" four times to the "source-value" key of the "species" property key, we can do so by issuing:

```
kim property modify 2 key species source-value 1 Al
kim property modify 2 key species source-value 2 Al
kim property modify 2 key species source-value 3 Al
kim property modify 2 key species source-value 4 Al
```

Note

No declaration of the number of elements in this array was given; *kim property modify* will automatically handle memory management to allow an arbitrary number of elements to be added to the array.

Note

In the event that *kim property modify* is used to set the value of an array index without having set the values of all lesser indices, they will be assigned default values based on the data type associated with the key in the map:

Data type	Default value
int	0
float	0.0
string	""
file	""

For example, doing the following:

```
kim property create 2 cohesive-energy-relation-cubic-crystal
kim property modify 2 key species source-value 4 Al
```

will result in the "source-value" key in the map for the property key "species" being assigned the array ["", "", "", "Al"].

For convenience, the index argument provided may refer to an inclusive range of indices by specifying two integers separated by a colon (the first integer must be less than or equal to the second integer, and no whitespace should be included). Thus, the snippet above could equivalently be written:

```
kim property modify 2 key species source-value 1:4 Al Al Al Al
```

Calling this command with a non-positive index, e.g. kim property modify 2 key species source-value 0 Al, or an incorrect number of input arguments, e.g. kim property modify 2 key species source-value 1:4 Al Al, will result in an error.

As an example of modifying multidimensional arrays, consider the "basis-atoms" key in the cohesive-energy-relation-cubic-crystal property definition. This is a two-dimensional array containing the fractional coordinates of atoms in the unit cell of the cubic crystal. In the case of, e.g. a conventional fcc unit cell, the "source-value" key in the map associated with this key should be assigned the following value:

```
[[0.0, 0.0, 0.0],
 [0.5, 0.5, 0.0],
 [0.5, 0.0, 0.5],
 [0.0, 0.5, 0.5]]
```

While each of the twelve components could be set individually, we can instead set each row at a time using colon notation:

```
kim property modify 2 key basis-atom-coordinates source-value 1 1:3 0.0 0.0 0.0
kim property modify 2 key basis-atom-coordinates source-value 2 1:3 0.5 0.5 0.0
kim property modify 2 key basis-atom-coordinates source-value 3 1:3 0.5 0.0 0.5
kim property modify 2 key basis-atom-coordinates source-value 4 1:3 0.0 0.5 0.5
```

Where the first index given refers to a row and the second index refers to a column. We could, instead, choose to set each column at a time like so:

```
kim property modify 2 key basis-atom-coordinates source-value 1:4 1 0.0 0.5 0.5 0.0 &
                    key basis-atom-coordinates source-value 1:4 2 0.0 0.5 0.0 0.5 &
                    key basis-atom-coordinates source-value 1:4 3 0.0 0.0 0.5 0.5
```

Note

Multiple calls of *kim property modify* made for the same instance ID can be combined into a single invocation, meaning the following are both valid:

```
kim property modify 2 key basis-atom-coordinates source-value 1 1:3 0.0 0.0 0.0 &
                    key basis-atom-coordinates source-value 2 1:3 0.5 0.5 0.0 &
                    key basis-atom-coordinates source-value 3 1:3 0.5 0.0 0.5 &
                    key basis-atom-coordinates source-value 4 1:3 0.0 0.5 0.5
```

```
kim property modify 2 key short-name source-value 1 fcc                    &
                    key species source-value 1:4 Al Al Al Al               &
                    key a source-value 1:5 3.9149 4.0000 4.032 4.0817 4.1602  &
                        source-unit angstrom                               &
                        digits 5                                           &
                    key basis-atom-coordinates source-value 1 1:3 0.0 0.0 0.0 &
                    key basis-atom-coordinates source-value 2 1:3 0.5 0.5 0.0 &
                    key basis-atom-coordinates source-value 3 1:3 0.5 0.0 0.5 &
                    key basis-atom-coordinates source-value 4 1:3 0.0 0.5 0.5
```

Note

For multidimensional arrays, only one colon-separated range is allowed in the index listing. Therefore,

```
kim property modify 2 key basis-atom-coordinates 1 1:3 0.0 0.0 0.0
```

is valid but

```
kim property modify 2 key basis-atom-coordinates 1:2 1:3 0.0 0.0 0.0 0.0 0.0 0.0
```

is not.

Note

After one sets a value in a map with the *kim property modify* command, additional calls will overwrite the previous value.

Remove

```
kim property remove instance_id key key_name
```

The *kim property remove* command can be used to remove a property key from a property instance. For example,

```
kim property remove 2 key basis-atom-coordinates
```

Destroy

```
kim property destroy instance_id
```

The *kim property destroy* command deletes a previously created property instance ID. For example,

```
kim property destroy 2
```

Note

If this command is called with an instance ID that does not exist, no error is raised.

Dump

The *kim property dump* command can be used to write the content of all currently defined property instances to a file:

```
kim property dump file
```

For example,

```
kim property dump results.edn
```

Note

Issuing the *kim property dump* command clears all existing property instances from memory.

1.44.8 Citation of OpenKIM IMs

When publishing results obtained using OpenKIM IMs researchers are requested to cite the OpenKIM project *(Tadmor)*, KIM API *(Elliott)*, and the specific IM codes used in the simulations, in addition to the relevant scientific references for the IM. The citation format for an IM is displayed on its page on OpenKIM along with the corresponding BibTex file, and is automatically added to the LAMMPS citation reminder.

Citing the IM software (KIM infrastructure and specific PM or SM codes) used in the simulation gives credit to the researchers who developed them and enables open source efforts like OpenKIM to function.

1.44.9 Restrictions

The *kim* command is part of the KIM package. It is only enabled if LAMMPS is built with that package. A requirement for the KIM package, is the KIM API library that must be downloaded from the OpenKIM website and installed before LAMMPS is compiled. When installing LAMMPS from binary, the kim-api package is a dependency that is automatically downloaded and installed. The *kim query* command requires the *libcurl* library to be installed. The *kim property* command requires *Python* 3.6 or later and the *kim-property* python package to be installed. See the KIM section of the *Packages details* for details.

Furthermore, when using *kim* command to run KIM SMs, any packages required by the native potential being used or other commands or fixes that it invokes must be installed.

1.44.10 Related commands

pair_style kim

(Tadmor) Tadmor, Elliott, Sethna, Miller and Becker, JOM, 63, 17 (2011). doi: https://doi.org/10.1007/s11837-011-0102-6

(Elliott) Elliott, Tadmor and Bernstein, https://openkim.org/kim-api (2011) doi: https://doi.org/10.25950/FF8F563A

1.45 kspace_modify command

1.45.1 Syntax

```
kspace_modify keyword value ...
```

- one or more keyword/value pairs may be listed
- keyword = *collective* or *compute* or *cutoff/adjust* or *diff* or *disp/auto* or *fftbench* or *force/disp/kspace* or *force/disp/real* or *force* or *gewald/disp* or *gewald* or *kmax/ewald* or *mesh* or *minorder* or *mix/disp* or *order/disp* or *order* or *overlap* or *scafacos* or *slab* or *splittol* or *wire*

  collective value = yes or no
  compute value = yes or no
  cutoff/adjust value = yes or no
  diff value = ad or ik = 2 or 4 FFTs for PPPM in smoothed or non-smoothed mode
  disp/auto value = yes or no
  fftbench value = yes or no
  force/disp/real value = accuracy (force units)
  force/disp/kspace value = accuracy (force units)

force value = accuracy (force units)
gewald value = rinv (1/distance units)
  rinv = G-ewald parameter for Coulombics
gewald/disp value = rinv (1/distance units)
  rinv = G-ewald parameter for dispersion
kmax/ewald value = kx ky kz
  kx,ky,kz = number of Ewald sum kspace vectors in each dimension
mesh value = x y z
  x,y,z = grid size in each dimension for long-range Coulombics
mesh/disp value = x y z
  x,y,z = grid size in each dimension for 1/r^6 dispersion
minorder value = M
  M = min allowed extent of Gaussian when auto-adjusting to minimize grid communication
mix/disp value = pair or geom or none
order value = N
  N = extent of Gaussian for PPPM or MSM mapping of charge to grid
order/disp value = N
  N = extent of Gaussian for PPPM mapping of dispersion term to grid
overlap = yes or no = whether the grid stencil for PPPM is allowed to overlap into more than the nearest-neighbor processor
pressure/scalar value = yes or no
scafacos values = option value1 value2 ...
  option = tolerance
    value = energy or energy_rel or field or field_rel or potential or potential_rel
  option = fmm_tuning
    value = 0 or 1
slab value = volfactor or nozforce
  volfactor = ratio of the total extended volume used in the
    2d approximation compared with the volume of the simulation domain
  nozforce turns off kspace forces in the z direction
splittol value = tol
  tol = relative size of two eigenvalues (see discussion below)
wire value = volfactor (available with ELECTRODE package)
  volfactor = ratio of the total extended dimension used in the 1d
    approximation compared with the dimension of the simulation domain

1.45.2 Examples

```
kspace_modify mesh 24 24 30 order 6
kspace_modify slab 3.0
kspace_modify scafacos tolerance energy
```

1.45.3 Description

Set parameters used by the kspace solvers defined by the *kspace_style* command. Not all parameters are relevant to all kspace styles.

The *collective* keyword applies only to PPPM. It is set to *no* by default, except on IBM BlueGene machines. If this option is set to *yes*, LAMMPS will use MPI collective operations to remap data for 3d-FFT operations instead of the default point-to-point communication. This is faster on IBM BlueGene machines, and may also be faster on other machines if they have an efficient implementation of MPI collective operations and adequate hardware.

The *compute* keyword allows Kspace computations to be turned off, even though a *kspace_style* is defined. This is not useful for running a real simulation, but can be useful for debugging purposes or for computing only partial forces that do not include the Kspace contribution. You can also do this by simply not defining a *kspace_style*, but a Kspace-compatible *pair_style* requires a kspace style to be defined. This keyword gives you that option.

The *cutoff/adjust* keyword applies only to MSM. If this option is turned on, the Coulombic cutoff will be automatically adjusted at the beginning of the run to give the desired estimated error. Other cutoffs such as LJ will not be affected. If the grid is not set using the *mesh* command, this command will also attempt to use the optimal grid that minimizes cost using an estimate given by *(Hardy)*. Note that this cost estimate is not exact, somewhat experimental, and still may not yield the optimal parameters.

The *diff* keyword specifies the differentiation scheme used by the PPPM method to compute forces on particles given electrostatic potentials on the PPPM mesh. The *ik* approach is the default for PPPM and is the original formulation used in *(Hockney)*. It performs differentiation in Kspace, and uses 3 FFTs to transfer each component of the computed fields back to real space for total of 4 FFTs per timestep.

The analytic differentiation *ad* approach uses only 1 FFT to transfer information back to real space for a total of 2 FFTs per timestep. It then performs analytic differentiation on the single quantity to generate the 3 components of the electric field at each grid point. This is sometimes referred to as "smoothed" PPPM. This approach requires a somewhat larger PPPM mesh to achieve the same accuracy as the *ik* method. Currently, only the *ik* method (default) can be used for a triclinic simulation cell with PPPM. The *ad* method is always used for MSM.

Note

Currently, not all PPPM styles support the *ad* option. Support for those PPPM variants will be added later.

The *disp/auto* option controls whether the pppm/disp is allowed to generate PPPM parameters automatically. If set to *no*, parameters have to be specified using the *gewald/disp*, *mesh/disp*, *force/disp/real* or *force/disp/kspace* keywords, or the code will stop with an error message. When this option is set to *yes*, the error message will not appear and the simulation will start. For a typical application, using the automatic parameter generation will provide simulations that are either inaccurate or slow. Using this option is thus not recommended. For guidelines on how to obtain good parameters, see the *long-range dispersion howto* discussion.

The *fftbench* keyword applies only to PPPM. It is off by default. If this option is turned on, LAMMPS will perform a short FFT benchmark computation and report its timings, and will thus finish some seconds later than it would if this option were off.

The *force/disp/real* and *force/disp/kspace* keywords set the force accuracy for the real and reciprocal space computations for the dispersion part of pppm/disp. As shown in *(Isele-Holder)*, optimal performance and accuracy in the results is obtained when these values are different.

The *force* keyword overrides the relative accuracy parameter set by the *kspace_style* command with an absolute accuracy. The accuracy determines the RMS error in per-atom forces calculated by the long-range solver and is thus specified in force units. A negative value for the accuracy setting means to use the relative accuracy parameter. The accuracy setting is used in conjunction with the pairwise cutoff to determine the number of K-space vectors for style *ewald*, the FFT grid size for style *pppm*, or the real space grid size for style *msm*.

The *gewald* keyword sets the value of the Ewald or PPPM G-ewald parameter for charge as *rinv* in reciprocal distance units. Without this setting, LAMMPS chooses the parameter automatically as a function of cutoff, precision, grid spacing, etc. This means it can vary from one simulation to the next which may not be desirable for matching a KSpace solver to a pre-tabulated pairwise potential. This setting can also be useful if Ewald or PPPM fails to choose a good grid spacing and G-ewald parameter automatically. If the value is set to 0.0, LAMMPS will choose the G-ewald parameter automatically. MSM does not use the *gewald* parameter.

The *gewald/disp* keyword sets the value of the Ewald or PPPM G-ewald parameter for dispersion as *rinv* in reciprocal distance units. It has the same meaning as the *gewald* setting for Coulombics.

The *kmax/ewald* keyword sets the number of kspace vectors in each dimension for kspace style *ewald*. The three values must be positive integers, or else (0,0,0), which unsets the option. When this option is not set, the Ewald sum scheme chooses its own kspace vectors, consistent with the user-specified accuracy and pairwise cutoff. In any case, if kspace style *ewald* is invoked, the values used are printed to the screen and the log file at the start of the run.

The *mesh* keyword sets the grid size for kspace style *pppm* or *msm*. In the case of PPPM, this is the FFT mesh, and each dimension must be factorizable into powers of 2, 3, and 5. In the case of MSM, this is the finest scale real-space mesh, and each dimension must be factorizable into powers of 2. When this option is not set, the PPPM or MSM solver chooses its own grid size, consistent with the user-specified accuracy and pairwise cutoff. Values for x,y,z of 0,0,0 unset the option.

The *mesh/disp* keyword sets the grid size for kspace style *pppm/disp*. This is the FFT mesh for long-range dispersion and ach dimension must be factorizable into powers of 2, 3, and 5. When this option is not set, the PPPM solver chooses its own grid size, consistent with the user-specified accuracy and pairwise cutoff. Values for x,y,z of 0,0,0 unset the option.

The *minorder* keyword allows LAMMPS to reduce the *order* setting if necessary to keep the communication of ghost grid point limited to exchanges between nearest-neighbor processors. See the discussion of the *overlap* keyword for details. If the *overlap* keyword is set to *yes*, which is the default, this is never needed. If it set to *no* and overlap occurs, then LAMMPS will reduce the order setting, one step at a time, until the ghost grid overlap only extends to nearest neighbor processors. The *minorder* keyword limits how small the *order* setting can become. The minimum allowed value for PPPM is 2, which is the default. If *minorder* is set to the same value as *order* then no reduction is allowed, and LAMMPS will generate an error if the grid communication is non-nearest-neighbor and *overlap* is set to *no*. The *minorder* keyword is not currently supported in MSM.

The *mix/disp* keyword selects the mixing rule for the dispersion coefficients. With *pair*, the dispersion coefficients of unlike types are computed as indicated with *pair_modify*. With *geom*, geometric mixing is enforced on the dispersion coefficients in the kspace coefficients. When using the arithmetic mixing rule, this will speed-up the simulations but introduces some error in the force computations, as shown in *(Wennberg)*. With *none*, it is assumed that no mixing rule is applicable. Splitting of the dispersion coefficients will be performed as described in *(Isele-Holder)*.

This splitting can be influenced with the *splittol* keywords. Only the eigenvalues that are larger than tol compared to the largest eigenvalues are included. Using this keywords the original matrix of dispersion coefficients is approximated. This leads to faster computations, but the accuracy in the reciprocal space computations of the dispersion part is decreased.

The *order* keyword determines how many grid spacings an atom's charge extends when it is mapped to the grid in kspace style *pppm* or *msm*. The default for this parameter is 5 for PPPM and 8 for MSM, which means each charge spans 5 or 8 grid cells in each dimension, respectively. For the LAMMPS implementation of MSM, the order can range from 4 to 10 and must be even. For PPPM, the minimum allowed setting is 2 and the maximum allowed setting is 7. The larger the value of this parameter, the smaller that LAMMPS will set the grid size, to achieve the requested accuracy. Conversely, the smaller the order value, the larger the grid size will be. Note that there is an inherent trade-off involved: a small grid will lower the cost of FFTs or MSM direct sum, but a larger order parameter will increase the cost of interpolating charge/fields to/from the grid.

The PPPM order parameter may be reset by LAMMPS when it sets up the FFT grid if the implied grid stencil extends beyond the grid cells owned by neighboring processors. Typically this will only occur when small problems are run on large numbers of processors. A warning will be generated indicating the order parameter is being reduced to allow LAMMPS to run the problem. Automatic adjustment of the order parameter is not supported in MSM.

The *order/disp* keyword determines how many grid spacings an atom's dispersion term extends when it is mapped to the grid in kspace style *pppm/disp*. It has the same meaning as the *order* setting for Coulombics.

The *overlap* keyword can be used in conjunction with the *minorder* keyword with the PPPM styles to adjust the amount of communication that occurs when values on the FFT grid are exchanged between processors. This communication is distinct from the communication inherent in the parallel FFTs themselves, and is required because processors interpolate charge and field values using grid point values owned by neighboring processors (i.e. ghost point communication). If the *overlap* keyword is set to *yes* then this communication is allowed to extend beyond nearest-neighbor processors, e.g. when using lots of processors on a small problem. If it is set to *no* then the communication will be limited to nearest-neighbor processors and the *order* setting will be reduced if necessary, as explained by the *minorder* keyword discussion. The *overlap* keyword is always set to *yes* in MSM.

The *pressure/scalar* keyword applies only to MSM. If this option is turned on, only the scalar pressure (i.e. (Pxx + Pyy + Pzz)/3.0) will be computed, which can be used, for example, to run an isotropic barostat. Computing the full pressure tensor with MSM is expensive, and this option provides a faster alternative. The scalar pressure is computed using a relationship between the Coulombic energy and pressure *(Hummer)* instead of using the virial equation. This option cannot be used to access individual components of the pressure tensor, to compute per-atom virial, or with suffix kspace/pair styles of MSM, like OMP or GPU.

The *scafacos* keyword is used for settings that are passed to the ScaFaCoS library when using *kspace_style scafacos*.

The *tolerance* option affects how the *accuracy* specified with the *kspace_style* command is interpreted by ScaFaCoS. The following values may be used:

- energy = absolute accuracy in total Coulombic energy
- energy_rel = relative accuracy in total Coulombic energy
- potential = absolute accuracy in total Coulombic potential
- potential_rel = relative accuracy in total Coulombic potential
- field = absolute accuracy in electric field
- field_rel = relative accuracy in electric field

The values with suffix _rel indicate the tolerance is a relative tolerance; the other values impose an absolute tolerance on the given quantity. Absolute tolerance in this case means, that for a given quantity q and a given absolute tolerance of t_a the result should be between q-t_a and q+t_a. For a relative tolerance t_r the relative error should not be greater than t_r, i.e. abs(1 - (result/q)) < t_r. As a consequence of this, the tolerance type should be checked, when performing computations with a high absolute field / energy. E.g. if the total energy in the system is 1000000.0 an absolute tolerance of 1e-3 would mean that the result has to be between 999999.999 and 1000000.001, which would be equivalent to a relative tolerance of 1e-9.

The energy and energy_rel values, set a tolerance based on the total Coulombic energy of the system. The potential and potential_rel set a tolerance based on the per-atom Coulombic energy. The field and field_rel tolerance types set a tolerance based on the electric field values computed by ScaFaCoS. Since per-atom forces are derived from the per-atom electric field, this effectively sets a tolerance on the forces, similar to other LAMMPS KSpace styles, as explained on the *kspace_style* doc page.

Note that not all ScaFaCoS solvers support all tolerance types. These are the allowed values for each method:

- fmm = energy and energy_rel
- p2nfft = field (1d-,2d-,3d-periodic systems) or potential (0d-periodic)
- p3m = field
- ewald = field
- direct = has no tolerance tuning

If the tolerance type is not changed, the default values for the tolerance type are the first values in the above list, e.g. energy is the default tolerance type for the fmm solver.

The *fmm_tuning* option is only relevant when using the FMM method. It activates (value=1) or deactivates (value=0) an internal tuning mechanism for the FMM solver. The tuning operation runs sequentially and can be very time-consuming. Usually it is not needed for systems with a homogeneous charge distribution. The default for this option is therefore *0*. The FMM internal tuning is performed once, when the solver is set up.

The *slab* keyword allows an Ewald or PPPM solver to be used for a systems that are periodic in x,y but non-periodic in z - a *boundary* setting of "boundary p p f". This is done by treating the system as if it were periodic in z, but inserting empty volume between atom slabs and removing dipole inter-slab interactions so that slab-slab interactions are effectively turned off. The volfactor value sets the ratio of the extended dimension in z divided by the actual dimension in z. It must be a value >= 1.0. A value of 1.0 (the default) means the slab approximation is not used.

The recommended value for volfactor is 3.0. A larger value is inefficient; a smaller value introduces unwanted slab-slab interactions. The use of fixed boundaries in z means that the user must prevent particle migration beyond the initial z-bounds, typically by providing a wall-style fix. The methodology behind the *slab* option is explained in the paper by *(Yeh)*. The *slab* option is also extended to non-neutral systems *(Ballenegger)*.

An alternative slab option can be invoked with the *nozforce* keyword in lieu of the volfactor. This turns off all kspace forces in the z direction. The *nozforce* option is not supported by MSM. For MSM, any combination of periodic, non-periodic, or shrink-wrapped boundaries can be set using *boundary* (the slab approximation in not needed). The *slab*

keyword is not currently supported by Ewald or PPPM when using a triclinic simulation cell. The slab correction has also been extended to point dipole interactions *(Klapp)* in *kspace_style* *ewald/disp*, *ewald/dipole*, and *pppm/dipole*.

Note

If you wish to apply an electric field in the Z-direction, in conjunction with the *slab* keyword, you should do it by adding explicit charged particles to the +/- Z surfaces. If you do it via the *fix efield* command, it will not give the correct dielectric constant due to the Yeh/Berkowitz *(Yeh)* correction not being compatible with how *fix efield* works.

The *force/disp/real* and *force/disp/kspace* keywords set the force accuracy for the real and reciprocal space computations for the dispersion part of pppm/disp. As shown in *(Isele-Holder)*, optimal performance and accuracy in the results is obtained when these values are different.

The *disp/auto* option controls whether the pppm/disp is allowed to generate PPPM parameters automatically. If set to *no*, parameters have to be specified using the *gewald/disp*, *mesh/disp*, *force/disp/real* or *force/disp/kspace* keywords, or the code will stop with an error message. When this option is set to *yes*, the error message will not appear and the simulation will start. For a typical application, using the automatic parameter generation will provide simulations that are either inaccurate or slow. Using this option is thus not recommended. For guidelines on how to obtain good parameters, see the *Howto dispersion* doc page.

1.45.4 Restrictions

none

1.45.5 Related commands

kspace_style, *boundary*

1.45.6 Default

The option defaults are as follows:

- compute = yes
- cutoff/adjust = yes (MSM)
- diff = ik (PPPM)
- disp/auto = no
- fftbench = no (PPPM)
- force = -1.0
- force/disp/kspace = -1.0
- force/disp/real = -1.0
- gewald = gewald/disp = 0.0
- mesh = mesh/disp = 0 0 0

- minorder = 2
- mix/disp = pair
- order = 10 (MSM)
- order = order/disp = 5 (PPPM)
- order = order/disp = 7 (PPPM/intel)
- overlap = yes
- pressure/scalar = yes (MSM)
- slab = 1.0
- split = 0
- tol = 1.0e-6

For scafacos settings, the scafacos tolerance option depends on the method chosen, as documented above. The scafacos fmm_tuning default = 0.

(Hockney) Hockney and Eastwood, Computer Simulation Using Particles, Adam Hilger, NY (1989).

(Yeh) Yeh and Berkowitz, J Chem Phys, 111, 3155 (1999).

(Ballenegger) Ballenegger, Arnold, Cerda, J Chem Phys, 131, 094107 (2009).

(Klapp) Klapp, Schoen, J Chem Phys, 117, 8050 (2002).

(Hardy) David Hardy thesis: Multilevel Summation for the Fast Evaluation of Forces for the Simulation of Biomolecules, University of Illinois at Urbana-Champaign, (2006).

(Hummer) Hummer, Gronbech-Jensen, Neumann, J Chem Phys, 109, 2791 (1998)

(Isele-Holder) Isele-Holder, Mitchell, Hammond, Kohlmeyer, Ismail, J Chem Theory Comput, 9, 5412 (2013).

(Wennberg) Wennberg, Murtola, Hess, Lindahl, J Chem Theory Comput, 9, 3527 (2013).

1.46 kspace_style command

1.46.1 Syntax

```
kspace_style style value
```

- style = *none* or *ewald* or *ewald/dipole* or *ewald/dipole/spin* or *ewald/disp* or *ewald/disp/dipole* or *ewald/omp* or *ewald/electrode* or *pppm* or *pppm/cg* or *pppm/disp* or *pppm/tip4p* or *pppm/stagger* or *pppm/disp/tip4p* or *pppm/gpu* or *pppm/intel* or *pppm/disp/intel* or *pppm/kk* or *pppm/omp* or *pppm/cg/omp* or *pppm/disp/tip4p/omp* or *pppm/tip4p/omp* or *pppm/dielectric* or *pppm/disp/dielectric* or *pppm/electrode* or *pppm/electrode/intel* or *msm* or *msm/cg* or *msm/omp* or *msm/cg/omp* or *msm/dielectric* or *scafacos*

  none value = none
  ewald value = accuracy
    accuracy = desired relative error in forces
  ewald/dipole value = accuracy
    accuracy = desired relative error in forces
  ewald/dipole/spin value = accuracy
    accuracy = desired relative error in forces

ewald/disp value = accuracy
  accuracy = desired relative error in forces
ewald/disp/dipole value = accuracy
  accuracy = desired relative error in forces
ewald/omp value = accuracy
  accuracy = desired relative error in forces
ewald/electrode value = accuracy
  accuracy = desired relative error in forces
pppm value = accuracy
  accuracy = desired relative error in forces
pppm/cg values = accuracy (smallq)
  accuracy = desired relative error in forces
  smallq = cutoff for charges to be considered (optional) (charge units)
pppm/dipole value = accuracy
  accuracy = desired relative error in forces
pppm/dipole/spin value = accuracy
  accuracy = desired relative error in forces
pppm/disp value = accuracy
  accuracy = desired relative error in forces
pppm/tip4p value = accuracy
  accuracy = desired relative error in forces
pppm/disp/tip4p value = accuracy
  accuracy = desired relative error in forces
pppm/gpu value = accuracy
  accuracy = desired relative error in forces
pppm/intel value = accuracy
  accuracy = desired relative error in forces
pppm/disp/intel value = accuracy
  accuracy = desired relative error in forces
pppm/kk value = accuracy
  accuracy = desired relative error in forces
pppm/omp value = accuracy
  accuracy = desired relative error in forces
pppm/cg/omp values = accuracy (smallq)
  accuracy = desired relative error in forces
  smallq = cutoff for charges to be considered (optional) (charge units)
pppm/disp/omp value = accuracy
  accuracy = desired relative error in forces
pppm/tip4p/omp value = accuracy
  accuracy = desired relative error in forces
pppm/disp/tip4p/omp value = accuracy
  accuracy = desired relative error in forces
pppm/stagger value = accuracy
  accuracy = desired relative error in forces
pppm/dielectric value = accuracy
  accuracy = desired relative error in forces
pppm/disp/dielectric value = accuracy
  accuracy = desired relative error in forces
pppm/electrode value = accuracy
  accuracy = desired relative error in forces
pppm/electrode/intel value = accuracy
  accuracy = desired relative error in forces
msm value = accuracy
  accuracy = desired relative error in forces

msm/cg value = accuracy (smallq)
  accuracy = desired relative error in forces
  smallq = cutoff for charges to be considered (optional) (charge units)
msm/omp value = accuracy
  accuracy = desired relative error in forces
msm/cg/omp value = accuracy (smallq)
  accuracy = desired relative error in forces
  smallq = cutoff for charges to be considered (optional) (charge units)
msm/dielectric value = accuracy
  accuracy = desired relative error in forces
scafacos values = method accuracy
  method = fmm or p2nfft or p3m or ewald or direct
  accuracy = desired relative error in forces

1.46.2 Examples

```
kspace_style pppm 1.0e-4
kspace_style pppm/cg 1.0e-5 1.0e-6
kspace_style msm 1.0e-4
kspace_style scafacos fmm 1.0e-4
kspace_style none
```

Used in input scripts:

```
examples/peptide/in.peptide
```

1.46.3 Description

Define a long-range solver for LAMMPS to use each timestep to compute long-range Coulombic interactions or long-range $1/r^6$ interactions. Most of the long-range solvers perform their computation in K-space, hence the name of this command.

When such a solver is used in conjunction with an appropriate pair style, the cutoff for Coulombic or $1/r^N$ interactions is effectively infinite. If the Coulombic case, this means each charge in the system interacts with charges in an infinite array of periodic images of the simulation domain.

Note that using a long-range solver requires use of a matching *pair style* to perform consistent short-range pairwise calculations. This means that the name of the pair style contains a matching keyword to the name of the KSpace style, as in this table:

Pair style	KSpace style
coul/long	ewald or pppm
coul/msm	msm
lj/long or buck/long	disp (for dispersion)
tip4p/long	tip4p
dipole/long	dipole

The *ewald* style performs a standard Ewald summation as described in any solid-state physics text.

The *ewald/disp* style adds a long-range dispersion sum option for $1/r^6$ potentials and is useful for simulation of interfaces *(Veld)*. It also performs standard Coulombic Ewald summations, but in a more efficient manner than the *ewald*

style. The $1/r^6$ capability means that Lennard-Jones or Buckingham potentials can be used without a cutoff, i.e. they become full long-range potentials.

The *ewald/disp/dipole* style can also be used with point-dipoles, see *(Toukmaji)*.

The *ewald/dipole* style adds long-range standard Ewald summations for dipole-dipole interactions, see *(Toukmaji)*.

The *ewald/dipole/spin* style adds long-range standard Ewald summations for magnetic dipole-dipole interactions between magnetic spins.

The *pppm* style invokes a particle-particle particle-mesh solver *(Hockney)* which maps atom charge to a 3d mesh, uses 3d FFTs to solve Poisson's equation on the mesh, then interpolates electric fields on the mesh points back to the atoms. It is closely related to the particle-mesh Ewald technique (PME) *(Darden)* used in AMBER and CHARMM. The cost of traditional Ewald summation scales as $N^{\frac{3}{2}}$ where N is the number of atoms in the system. The PPPM solver scales as $N \log N$ due to the FFTs, so it is almost always a faster choice *(Pollock)*.

The *pppm/cg* style is identical to the *pppm* style except that it has an optimization for systems where most particles are uncharged. Similarly the *msm/cg* style implements the same optimization for *msm*. The optional *smallq* argument defines the cutoff for the absolute charge value which determines whether a particle is considered charged or not. Its default value is 1.0e-5.

The *pppm/dipole* style invokes a particle-particle particle-mesh solver for dipole-dipole interactions, following the method of *(Cerda)*.

The *pppm/dipole/spin* style invokes a particle-particle particle-mesh solver for magnetic dipole-dipole interactions between magnetic spins.

The *pppm/tip4p* style is identical to the *pppm* style except that it adds a charge at the massless fourth site in each TIP4P water molecule. It should be used with *pair styles* with a *tip4p/long* in their style name.

The *pppm/stagger* style performs calculations using two different meshes, one shifted slightly with respect to the other. This can reduce force aliasing errors and increase the accuracy of the method for a given mesh size. Or a coarser mesh can be used for the same target accuracy, which saves CPU time. However, there is a trade-off since FFTs on two meshes are now performed which increases the computation required. See *(Cerutti)*, *(Neelov)*, and *(Hockney)* for details of the method.

For high relative accuracy, using staggered PPPM allows the mesh size to be reduced by a factor of 2 in each dimension as compared to regular PPPM (for the same target accuracy). This can give up to a 4x speedup in the KSpace time (8x less mesh points, 2x more expensive). However, for low relative accuracy, the staggered PPPM mesh size may be essentially the same as for regular PPPM, which means the method will be up to 2x slower in the KSpace time (simply 2x more expensive). For more details and timings, see the *Speed tips* doc page.

Note

Using *pppm/stagger* may not give the same increase in the accuracy of energy and pressure as it does in forces, so some caution must be used if energy and/or pressure are quantities of interest, such as when using a barostat.

The *pppm/disp* and *pppm/disp/tip4p* styles add a mesh-based long-range dispersion sum option for 1/r^6 potentials *(Isele-Holder)*, similar to the *ewald/disp* style. The 1/r^6 capability means that Lennard-Jones or Buckingham potentials can be used without a cutoff, i.e. they become full long-range potentials.

For these styles, you will possibly want to adjust the default choice of parameters by using the *kspace_modify* command. This can be done by either choosing the Ewald and grid parameters, or by specifying separate accuracies for the real and kspace calculations. When not making any settings, the simulation will stop with an error message. Further information on the influence of the parameters and how to choose them is described in *(Isele-Holder)*, *(Isele-Holder2)* and the *Howto dispersion* doc page.

Note

All of the PPPM styles can be used with single-precision FFTs by using the compiler switch -DFFT_SINGLE for the FFT_INC setting in your low-level Makefile. This setting also changes some of the PPPM operations (e.g. mapping charge to mesh and interpolating electric fields to particles) to be performed in single precision. This option can speed-up long-range calculations, particularly in parallel or on GPUs. The use of the -DFFT_SINGLE flag is discussed on the *Build settings* doc page. MSM does not currently support the -DFFT_SINGLE compiler switch.

The *electrode* styles add methods that are required for the constant potential method implemented in *fix electrode/** . The styles *ewald/electrode*, *pppm/electrode* and *pppm/electrode/intel* are available. These styles do not support the *kspace_modify slab nozforce* command.

The *msm* style invokes a multi-level summation method MSM solver, *(Hardy)* or *(Hardy2)*, which maps atom charge to a 3d mesh, and uses a multi-level hierarchy of coarser and coarser meshes on which direct Coulomb solvers are done. This method does not use FFTs and scales as N. It may therefore be faster than the other K-space solvers for relatively large problems when running on large core counts. MSM can also be used for non-periodic boundary conditions and for mixed periodic and non-periodic boundaries.

MSM is most competitive versus Ewald and PPPM when only relatively low accuracy forces, about 1e-4 relative error or less accurate, are needed. Note that use of a larger Coulombic cutoff (i.e. 15 Angstroms instead of 10 Angstroms) provides better MSM accuracy for both the real space and grid computed forces.

Currently calculation of the full pressure tensor in MSM is expensive. Using the *kspace_modify pressure/scalar yes* command provides a less expensive way to compute the scalar pressure (Pxx + Pyy + Pzz)/3.0. The scalar pressure can be used, for example, to run an isotropic barostat. If the full pressure tensor is needed, then calculating the pressure at every timestep or using a fixed pressure simulation with MSM will cause the code to run slower.

The *scafacos* style is a wrapper on the ScaFaCoS Coulomb solver library which provides a variety of solver methods which can be used with LAMMPS. The paper by *(Sutman)* gives an overview of ScaFaCoS.

ScaFaCoS was developed by a consortium of German research facilities with a BMBF (German Ministry of Science and Education) funded project in 2009-2012. Participants of the consortium were the Universities of Bonn, Chemnitz, Stuttgart, and Wuppertal as well as the Forschungszentrum Juelich.

The library is available for download at "http://scafacos.de" or can be cloned from the git-repository "https://github.com/scafacos/scafacos.git".

In order to use this KSpace style, you must download and build the ScaFaCoS library, then build LAMMPS with the SCAFACOS package installed package which links LAMMPS to the ScaFaCoS library. See details on *this page*.

Note

Unlike other KSpace solvers in LAMMPS, ScaFaCoS computes all Coulombic interactions, both short- and long-range. Thus you should NOT use a Coulombic pair style when using kspace_style scafacos. This also means the total Coulombic energy (short- and long-range) will be tallied for *thermodynamic output* command as part of the *elong* keyword; the *ecoul* keyword will be zero.

Note

See the current restriction below about use of ScaFaCoS in LAMMPS with molecular charged systems or the TIP4P water model.

The specified *method* determines which ScaFaCoS algorithm is used. These are the ScaFaCoS methods currently available from LAMMPS:

- *fmm* = Fast Multi-Pole method
- *p2nfft* = FFT-based Coulomb solver
- *ewald* = Ewald summation
- *direct* = direct O(N^2) summation
- *p3m* = PPPM

We plan to support additional ScaFaCoS solvers from LAMMPS in the future. For an overview of the included solvers, refer to *(Sutmann)*

The specified *accuracy* is similar to the accuracy setting for other LAMMPS KSpace styles, but is passed to ScaFaCoS, which can interpret it in different ways for different methods it supports. Within the ScaFaCoS library the *accuracy* is treated as a tolerance level (either absolute or relative) for the chosen quantity, where the quantity can be either the Columic field values, the per-atom Columic energy or the total Columic energy. To select from these options, see the *kspace_modify scafacos accuracy* doc page.

The *kspace_modify scafacos* command also explains other ScaFaCoS options currently exposed to LAMMPS.

The specified *accuracy* determines the relative RMS error in per-atom forces calculated by the long-range solver. It is set as a dimensionless number, relative to the force that two unit point charges (e.g. 2 monovalent ions) exert on each other at a distance of 1 Angstrom. This reference value was chosen as representative of the magnitude of electrostatic forces in atomic systems. Thus an accuracy value of 1.0e-4 means that the RMS error will be a factor of 10000 smaller than the reference force.

The accuracy setting is used in conjunction with the pairwise cutoff to determine the number of K-space vectors for style *ewald* or the grid size for style *pppm* or *msm*.

Note that style *pppm* only computes the grid size at the beginning of a simulation, so if the length or triclinic tilt of the simulation cell increases dramatically during the course of the simulation, the accuracy of the simulation may degrade. Likewise, if the *kspace_modify slab* option is used with shrink-wrap boundaries in the z-dimension, and the box size changes dramatically in z. For example, for a triclinic system with all three tilt factors set to the maximum limit, the PPPM grid should be increased roughly by a factor of 1.5 in the y direction and 2.0 in the z direction as compared to the same system using a cubic orthogonal simulation cell. One way to handle this issue if you have a long simulation where the box size changes dramatically, is to break it into shorter simulations (multiple *run* commands). This works because the grid size is re-computed at the beginning of each run. Another way to ensure the described accuracy requirement is met is to run a short simulation at the maximum expected tilt or length, note the required grid size, and then use the *kspace_modify mesh* command to manually set the PPPM grid size to this value for the long run. The simulation then will be "too accurate" for some portion of the run.

RMS force errors in real space for *ewald* and *pppm* are estimated using equation 18 of *(Kolafa)*, which is also referenced as equation 9 of *(Petersen)*. RMS force errors in K-space for *ewald* are estimated using equation 11 of *(Petersen)*, which is similar to equation 32 of *(Kolafa)*. RMS force errors in K-space for *pppm* are estimated using equation 38 of *(Deserno)*. RMS force errors for *msm* are estimated using ideas from chapter 3 of *(Hardy)*, with equation 3.197 of particular note. When using *msm* with non-periodic boundary conditions, it is expected that the error estimation will be too pessimistic. RMS force errors for dipoles when using *ewald/disp* or *ewald/dipole* are estimated using equations 33 and 46 of *(Wang)*. The RMS force errors for *pppm/dipole* are estimated using the equations in *(Cerda)*.

See the *kspace_modify* command for additional options of the K-space solvers that can be set, including a *force* option for setting an absolute RMS error in forces, as opposed to a relative RMS error.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

Note

For the GPU package, the *pppm/gpu* style performs charge assignment and force interpolation calculations on the GPU. These processes are performed either in single or double precision, depending on whether the -DFFT_SINGLE setting was specified in your low-level Makefile, as discussed above. The FFTs themselves are still calculated on the CPU. If *pppm/gpu* is used with a GPU-enabled pair style, part of the PPPM calculation can be performed concurrently on the GPU while other calculations for non-bonded and bonded force calculation are performed on the CPU.

Note

For the KOKKOS package, the *pppm/kk* style performs charge assignment and force interpolation calculations, along with the FFTs themselves, on the GPU or (optionally) threaded on the CPU when using OpenMP and FFTW3. The specific FFT library is selected using the FFT_KOKKOS CMake parameter. See the *Build settings* doc page for how to select a 3rd-party FFT library.

1.46.4 Restrictions

Note that the long-range electrostatic solvers in LAMMPS assume conducting metal (tinfoil) boundary conditions for both charge and dipole interactions. Vacuum boundary conditions are not currently supported.

The *ewald/disp*, *ewald*, *pppm*, and *msm* styles support non-orthogonal (triclinic symmetry) simulation boxes. However, triclinic simulation cells may not yet be supported by all suffix versions of these styles.

Most of the base kspace styles are part of the KSPACE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The *msm/dielectric* and *pppm/dielectric* kspace styles are part of the DIELECTRIC package. They are only enabled if LAMMPS was built with that package **and** the KSPACE package. See the *Build package* page for more info.

For MSM, a simulation must be 3d and one can use any combination of periodic, non-periodic, but not shrink-wrapped boundaries (specified using the *boundary* command).

For Ewald and PPPM, a simulation must be 3d and periodic in all dimensions. The only exception is if the slab option is set with *kspace_modify*, in which case the xy dimensions must be periodic and the z dimension must be non-periodic.

The scafacos KSpace style will only be enabled if LAMMPS is built with the SCAFACOS package. See the *Build package* doc page for more info.

The use of ScaFaCos in LAMMPS does not yet support molecular charged systems where the short-range Coulombic interactions between atoms in the same bond/angle/dihedral are weighted by the *special_bonds* command. Likewise it does not support the "TIP4P water style" where a fictitious charge site is introduced in each water molecule. Finally, the methods *p3m* and *ewald* do not support computing the virial, so this contribution is not included.

1.46.5 Related commands

kspace_modify, *pair_style lj/cut/coul/long*, *pair_style lj/charmm/coul/long*, *pair_style lj/long/coul/long*, *pair_style buck/coul/long*

1.46.6 Default

```
kspace_style none
```

(Darden) Darden, York, Pedersen, J Chem Phys, 98, 10089 (1993).

(Deserno) Deserno and Holm, J Chem Phys, 109, 7694 (1998).

(Hockney) Hockney and Eastwood, Computer Simulation Using Particles, Adam Hilger, NY (1989).

(Kolafa) Kolafa and Perram, Molecular Simulation, 9, 351 (1992).

(Petersen) Petersen, J Chem Phys, 103, 3668 (1995).

(Wang) Wang and Holm, J Chem Phys, 115, 6277 (2001).

(Pollock) Pollock and Glosli, Comp Phys Comm, 95, 93 (1996).

(Cerutti) Cerutti, Duke, Darden, Lybrand, Journal of Chemical Theory and Computation 5, 2322 (2009)

(Neelov) Neelov, Holm, J Chem Phys 132, 234103 (2010)

(Veld) In 't Veld, Ismail, Grest, J Chem Phys, 127, 144711 (2007).

(Toukmaji) Toukmaji, Sagui, Board, and Darden, J Chem Phys, 113, 10913 (2000).

(Isele-Holder) Isele-Holder, Mitchell, Ismail, J Chem Phys, 137, 174107 (2012).

(Isele-Holder2) Isele-Holder, Mitchell, Hammond, Kohlmeyer, Ismail, J Chem Theory Comput 9, 5412 (2013).

(Hardy) David Hardy thesis: Multilevel Summation for the Fast Evaluation of Forces for the Simulation of Biomolecules, University of Illinois at Urbana-Champaign, (2006).

(Hardy2) Hardy, Stone, Schulten, Parallel Computing, 35, 164-177 (2009).

(Sutmann) Sutmann, Arnold, Fahrenberger, et. al., Physical review / E 88(6), 063308 (2013)

(Cerda) Cerda, Ballenegger, Lenz, Holm, J Chem Phys 129, 234104 (2008)

(Sutmann) G. Sutmann. ScaFaCoS - a Scalable library of Fast Coulomb Solvers for particle Systems. In Bajaj, Zavattieri, Koslowski, Siegmund, Proceedings of the Society of Engineering Science 51st Annual Technical Meeting. 2014.

1.47 label command

1.47.1 Syntax

```
label ID
```

- ID = string used as label name

1.47.2 Examples

```
label xyz
label loop
```

1.47.3 Description

Label this line of the input script with the chosen ID. Unless a jump command was used previously, this does nothing. But if a *jump* command was used with a label argument to begin invoking this script file, then all command lines in the script prior to this line will be ignored. I.e. execution of the script will begin at this line. This is useful for looping over a section of the input script as discussed in the *jump* command.

1.47.4 Restrictions

none

1.47.5 Related commands

jump, *next*

1.47.6 Default

none

1.48 labelmap command

1.48.1 Syntax

```
labelmap option args
```

- *option* = *atom* or *bond* or *angle* or *dihedral* or *improper* or *clear* or *write*

  clear = no args
  write arg = filename
  atom or bond or angle or dihedral or improper
    args = list of one or more numeric-type/type-label pairs

1.48.2 Examples

```
labelmap atom 1 c1 2 hc 3 cp 4 nt
labelmap atom 3 carbon 4 'c3"' 5 "c1'" 6 "c#"
labelmap atom $(label2type(atom,carbon)) C  # change type label from 'carbon' to 'C'
labelmap clear
labelmap write mymap.include
labelmap bond 1 carbonyl 2 nitrile 3 """ c1'-c2" """
```

1.48.3 Description

Added in version 15Sep2022.

Define alphanumeric type labels to associate with one or more numeric atom, bond, angle, dihedral or improper types. A collection of type labels for all atom types, bond types, etc. is stored as a label map.

The label map can also be defined by the *read_data* command when it reads these sections in a data file: Atom Type Labels, Bond Type Labels, etc. See the *Howto type labels* doc page for a general discussion of how type labels can be used. See *(Gissinger)* for a discussion of the type label implementation in LAMMPS and its uses.

Valid type labels can contain any alphanumeric character, but must not start with a number, a '#', or a '*' character. They can contain other standard ASCII characters such as angular or square brackets '<' and '>' or '[' and ']', parenthesis '(' and ')', dash '-', underscore '_', plus '+' and equals '=' signs and more. They must not contain blanks or any other whitespace. Note that type labels must be put in single or double quotation marks if they contain the '#' character or if they contain a double (") or single quotation mark ('). If the label contains both a single and a double quotation mark, then triple quotation (""") must be used. When enclosing a type label with quotation marks, the LAMMPS input parser may require adding leading or trailing blanks around the type label so it can identify the enclosing quotation marks. Those blanks will be removed when defining the label.

A *labelmap* command can only modify the label map for one type-kind (atom types, bond types, etc). Any number of numeric-type/type-label pairs may follow. If a type label already exists for the same numeric type, it will be overwritten. Type labels must be unique; assigning the same type label to multiple numeric types within the same type-kind is not allowed. When reading and writing data files, it is required that there is a label defined for *every* numeric type within a given type-kind in order to write out the type label section for that type-kind.

The *clear* option resets the label map and thus discards all previous settings.

The *write* option takes a filename as argument and writes the current label mappings to a file as a sequence of *labelmap* commands, so the file can be copied into a new LAMMPS input file or read in using the *include* command.

1.48.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

Label maps are currently not supported when using the KOKKOS package.

1.48.5 Related commands

read_data, *write_data*, *molecule*, *fix bond/react*

1.48.6 Default

none

(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).

1.49 lattice command

1.49.1 Syntax

```
lattice style scale keyword values ...
```

- style = *none* or *sc* or *bcc* or *fcc* or *hcp* or *diamond* or *sq* or *sq2* or *hex* or *custom*
- scale = scale factor between lattice and simulation box

  scale = reduced density rho* (for LJ units)
  scale = lattice constant in distance units (for all other units)

- zero or more keyword/value pairs may be appended
- keyword = *origin* or *orient* or *spacing* or *a1* or *a2* or *a3* or *basis* or *triclinic/general*

  origin values = x y z
    x,y,z = fractions of a unit cell (0 <= x,y,z < 1)
  orient values = dim i j k
    dim = x or y or z
    i,j,k = integer lattice directions
  spacing values = dx dy dz
    dx,dy,dz = lattice spacings in the x,y,z box directions
  a1,a2,a3 values = x y z
    x,y,z = primitive vector components that define unit cell
  basis values = x y z
    x,y,z = fractional coords of a basis atom (0 <= x,y,z < 1)
  triclinic/general values = no values

1.49.2 Examples

```
lattice fcc 3.52
lattice hex 0.85
lattice sq 0.8 origin 0.0 0.5 0.0 orient x 1 1 0 orient y -1 1 0
lattice custom 3.52 a1 1.0 0.0 0.0 a2 0.5 1.0 0.0 a3 0.0 0.0 0.5 &
                basis 0.0 0.0 0.0 basis 0.5 0.5 0.5 triclinic/general
lattice none 2.0
```

1.49.3 Description

Define a lattice for use by other commands. In LAMMPS, a lattice is simply a set of points in space, determined by a unit cell with basis atoms, that is replicated infinitely in all dimensions. The arguments of the lattice command can be used to define a wide variety of crystallographic lattices.

A lattice is used by LAMMPS in two ways. First, the *create_atoms* command creates atoms on the lattice points inside the simulation box. Note that the *create_atoms* command allows different atom types to be assigned to different basis atoms of the lattice. Second, the lattice spacing in the x,y,z dimensions implied by the lattice, can be used by other commands as distance units (e.g. *create_box*, *region* and *velocity*), which are often convenient to use when the underlying problem geometry is atoms on a lattice.

The lattice style must be consistent with the dimension of the simulation - see the *dimension* command. Styles *sc* or *bcc* or *fcc* or *hcp* or *diamond* are for 3d problems. Styles *sq* or *sq2* or *hex* are for 2d problems. Style *custom* can be used for either 2d or 3d problems.

A lattice consists of a unit cell, a set of basis atoms within that cell, and a set of transformation parameters (scale, origin, orient) that map the unit cell into the simulation box. The vectors a1,a2,a3 are the edge vectors of the unit cell. This is the nomenclature for "primitive" vectors in solid-state crystallography, but in LAMMPS the unit cell they determine does not have to be a "primitive cell" of minimum volume.

Note that the lattice command can be used multiple times in an input script. Each time it is invoked, the lattice attributes are re-defined and are used for all subsequent commands (that use lattice attributes). For example, a sequence of lattice, *region*, and *create_atoms* commands can be repeated multiple times to build a poly-crystalline model with different geometric regions populated with atoms in different lattice orientations.

A lattice of style *none* does not define a unit cell and basis set, so it cannot be used with the *create_atoms* command. However it does define a lattice spacing via the specified scale parameter. As explained above the lattice spacings in x,y,z can be used by other commands as distance units. No additional keyword/value pairs can be specified for the *none* style. By default, a "lattice none 1.0" is defined, which means the lattice spacing is the same as one distance unit, as defined by the *units* command.

Lattices of style *sc*, *fcc*, *bcc*, and *diamond* are 3d lattices that define a cubic unit cell with edge length = 1.0. This means a1 = 1 0 0, a2 = 0 1 0, and a3 = 0 0 1. Style *hcp* has a1 = 1 0 0, a2 = 0 sqrt(3) 0, and a3 = 0 0 sqrt(8/3). The placement of the basis atoms within the unit cell are described in any solid-state physics text. A *sc* lattice has 1 basis atom at the lower-left-bottom corner of the cube. A *bcc* lattice has 2 basis atoms, one at the corner and one at the center of the cube. A *fcc* lattice has 4 basis atoms, one at the corner and 3 at the cube face centers. A *hcp* lattice has 4 basis atoms, two in the z = 0 plane and 2 in the z = 0.5 plane. A *diamond* lattice has 8 basis atoms.

Lattices of style *sq* and *sq2* are 2d lattices that define a square unit cell with edge length = 1.0. This means a1 = 1 0 0 and a2 = 0 1 0. A *sq* lattice has 1 basis atom at the lower-left corner of the square. A *sq2* lattice has 2 basis atoms, one at the corner and one at the center of the square. A *hex* style is also a 2d lattice, but the unit cell is rectangular, with a1 = 1 0 0 and a2 = 0 sqrt(3) 0. It has 2 basis atoms, one at the corner and one at the center of the rectangle.

A lattice of style *custom* allows you to specify a1, a2, a3, and a list of basis atoms to put in the unit cell. By default, a1 and a2 and a3 are 3 orthogonal unit vectors (edges of a unit cube). But you can specify them to be of any length and non-orthogonal to each other, so that they describe a tilted parallelepiped. Via the *basis* keyword you add atoms, one at a time, to the unit cell. Its arguments are fractional coordinates (0.0 <= x,y,z < 1.0). For 2d simulations, the fractional z coordinate for any basis atom must be 0.0.

The position vector x of a basis atom within the unit cell is a linear combination of the unit cell's 3 edge vectors, i.e. x = bx a1 + by a2 + bz a3, where bx,by,bz are the 3 values specified for the *basis* keyword.

This subsection discusses the arguments that determine how the idealized unit cell is transformed into a lattice of points within the simulation box.

The *scale* argument determines how the size of the unit cell will be scaled when mapping it into the simulation box. I.e. it determines a multiplicative factor to apply to the unit cell, to convert it to a lattice of the desired size and distance units in the simulation box. The meaning of the *scale* argument depends on the *units* being used in your simulation.

For all unit styles except *lj*, the scale argument is specified in the distance units defined by the unit style. For example, in *real* or *metal* units, if the unit cell is a unit cube with edge length 1.0, specifying scale = 3.52 would create a cubic lattice with a spacing of 3.52 Angstroms. In *cgs* units, the spacing would be 3.52 cm.

For unit style *lj*, the scale argument is the Lennard-Jones reduced density, typically written as rho*. LAMMPS converts this value into the multiplicative factor via the formula "factor^dim = rho/rho*", where rho = N/V with V = the volume of the lattice unit cell and N = the number of basis atoms in the unit cell (described below), and dim = 2 or 3 for the dimensionality of the simulation. Effectively, this means that if LJ particles of size sigma = 1.0 are used in the simulation, the lattice of particles will be at the desired reduced density.

The *origin* option specifies how the unit cell will be shifted or translated when mapping it into the simulation box. The x,y,z values are fractional values (0.0 <= x,y,z < 1.0) meaning shift the lattice by a fraction of the lattice spacing in each dimension. The meaning of "lattice spacing" is discussed below. For 2d simulations, the *origin* z value must be 0.0.

The *orient* option specifies how the unit cell will be rotated when mapping it into the simulation box. The *dim* argument is one of the 3 coordinate axes in the simulation box. The other 3 arguments are the crystallographic direction in the lattice that you want to orient along that axis, specified as integers. E.g. "orient x 2 1 0" means the x-axis in the simulation box will be the [210] lattice direction, and similarly for y and z. The 3 lattice directions you specify do not have to be unit vectors, but they must be mutually orthogonal and obey the right-hand rule, i.e. (X cross Y) points in the Z direction. For 2d simulations, the *orient* x and y vectors must define 0 for their 3rd component. Similarly the *orient* z vector must define 0 for its 1st and 2nd components.

Note

The preceding paragraph describing lattice directions is only valid for orthogonal cubic unit cells (or square in 2d). If you are using a *hcp* or *hex* lattice or the more general lattice style *custom* with non-orthogonal a1,a2,a3 vectors, then you should think of the 3 *orient* vectors as creating a 3x3 rotation matrix which is applied to a1,a2,a3 to rotate the original unit cell to a new orientation in the simulation box.

The *triclinic/general* option specifies that the defined lattice is for use with a general triclinic simulation box, as opposed to an orthogonal or restricted triclinic box. The *Howto triclinic* doc page explains all 3 kinds of simulation boxes LAMMPS supports.

If this option is specified, a *custom* lattice style must be used. The *a1*, *a2*, *a3* vectors should define the edge vectors of a single unit cell of the lattice with one or more basis atoms. They edge vectors can be arbitrary so long as they are non-zero, distinct, and not co-planar. In addition, they must define a right-handed system, such that (*a1* cross *a2*) points in the direction of *a3*. Note that a left-handed system can be converted to a right-handed system by simply swapping the order of any pair of the *a1*, *a2*, *a3* vectors. For 2d simulations, the *a3* vector must be specified as (0.0,0.0,1.0), which is its default value.

If this option is used, the *origin* and *orient* settings must have their default values. Namely (0.0,0.0,0.0) for the *origin* and (100), (010), (001) for the *orient* vectors.

The *create_box* command can be used to create a general triclinic box that replicates the *a1*, *a2*, *a3* unit cell vectors in each direction to create the 3 arbitrary edge vectors of the overall simulation box. It requires a lattice with the *triclinic/general* option.

Likewise, the *create_atoms* command can be used to add atoms (or molecules) to a general triclinic box which lie on the lattice points defined by *a1*, *a2*, *a3* and the unit cell basis atoms. To do this, it also requires a lattice with the *triclinic/general* option.

Note

LAMMPS allows specification of general triclinic lattices and simulation boxes as a convenience for users who may be converting data from solid-state crystallographic representations or from DFT codes for input to LAMMPS. However, as explained on the *Howto_triclinic* doc page, internally, LAMMPS only uses restricted triclinic simulation boxes. This means the box and per-atom information (e.g. coordinates, velocities) defined by the *create_box* and *create_atoms* commands are converted from general to restricted triclinic form when the two commands are invoked. It also means that any other commands which use lattice spacings from this command (e.g. the region command), will be operating on a restricted triclinic simulation box, even if the *triclinic/general* option was used to define the lattice. See the next section for details.

Several LAMMPS commands have the option to use distance units that are inferred from "lattice spacings" in the x,y,z box directions. E.g. the *region* command can create a block of size 10x20x20, where 10 means 10 lattice spacings in the x direction.

Note

Though they are called lattice spacings, all the commands that have a "units lattice" option, simply use the 3 values as scale factors on the distance units defined by the *units* command. Thus if you do not like the lattice spacings computed by LAMMPS (e.g. for a non-orthogonal or rotated unit cell), you can define the 3 values to be whatever you wish, via the *spacing* option.

If the *spacing* option is not specified, the lattice spacings are computed by LAMMPS in the following way. A unit cell of the lattice is mapped into the simulation box (scaled and rotated), so that it now has (perhaps) a modified size and orientation. The lattice spacing in X is defined as the difference between the min/max extent of the x coordinates of the 8 corner points of the modified unit cell (4 in 2d). Similarly, the Y and Z lattice spacings are defined as the difference in the min/max of the y and z coordinates.

Note

If the *triclinic/general* option is specified, the unit cell defined by *a1*, *a2*, *a3* edge vectors is first converted to a restricted triclinic orientation, which is a rotation operation. The min/max extent of the 8 corner points is then determined, as described in the preceding paragraph, to set the lattice spacings. As explained for the *triclinic/general* option above, this is because any use of the lattice spacings by other commands will be for a restricted triclinic simulation box, not a general triclinic box.

Note that if the unit cell is orthogonal with axis-aligned edges (no rotation via the *orient* keyword), then the lattice spacings in each dimension are simply the scale factor (described above) multiplied by the length of a1,a2,a3. Thus a *hex* style lattice with a scale factor of 3.0 Angstroms, would have a lattice spacing of 3.0 in x and 3*sqrt(3.0) in y.

Note

For non-orthogonal unit cells and/or when a rotation is applied via the *orient* keyword, then the lattice spacings computed by LAMMPS are typically less intuitive. In particular, in these cases, there is no guarantee that a particular lattice spacing is an integer multiple of the periodicity of the lattice in that direction. Thus, if you create an orthogonal periodic simulation box whose size in a dimension is a multiple of the lattice spacing, and then fill it with atoms via the *create_atoms* command, you will NOT necessarily create a periodic system. I.e. atoms may overlap incorrectly at the faces of the simulation box.

The *spacing* option sets the 3 lattice spacings directly. All must be non-zero (use 1.0 for dz in a 2d simulation). The specified values are multiplied by the multiplicative factor described above that is associated with the scale factor. Thus a spacing of 1.0 means one unit cell edge length independent of the scale factor. As mentioned above, this option can be useful if the spacings LAMMPS computes are inconvenient to use in subsequent commands, which can be the case for non-orthogonal or rotated lattices.

Note that whenever the lattice command is used, the values of the lattice spacings LAMMPS calculates are printed out. Thus their effect in commands that use the spacings should be decipherable.

Example commands for generating a Wurtzite crystal. The lattice constants approximate those of CdSe. The $\sqrt{3} \times 1$ orthorhombic supercell is used with the x, y, and z directions oriented along $[\bar{1}\bar{2}30]$, $[10\bar{1}0]$, and [0001], respectively.

```
variable a equal  4.34
variable b equal  $a*sqrt(3.0)
variable c equal  $a*sqrt(8.0/3.0)

variable third equal 1.0/3.0
variable five6 equal 5.0/6.0

lattice custom    1.0     &
       a1      $b       0.0     0.0     &
       a2      0.0      $a      0.0     &
       a3      0.0      0.0     $c      &
       basis   0.0      0.0     0.0     &
       basis   0.5      0.5     0.0     &
       basis   ${third} 0.0     0.5     &
       basis   ${five6} 0.5     0.5     &
       basis   0.0      0.0     0.625   &
       basis   0.5      0.5     0.625   &
       basis   ${third} 0.0     0.125   &
       basis   ${five6} 0.5     0.125

region myreg block 0 1 0 1 0 1
create_box      2 myreg
create_atoms    1 box     &
       basis   5       2       &
       basis   6       2       &
       basis   7       2       &
       basis   8       2
```

1.49.4 Restrictions

The *a1,a2,a3,basis* keywords can only be used with style *custom*.

1.49.5 Related commands

dimension, *create_atoms*, *region*

1.49.6 Default

```
lattice none 1.0
```

For other lattice styles, the option defaults are origin = 0.0 0.0 0.0, orient = x 1 0 0, orient = y 0 1 0, orient = z 0 0 1, a1 = 1 0 0, a2 = 0 1 0, and a3 = 0 0 1.

1.50 log command

1.50.1 Syntax

```
log file keyword
```

- file = name of new logfile
- keyword = *append* if output should be appended to logfile (optional)

1.50.2 Examples

```
log log.equil
log log.equil append
```

1.50.3 Description

This command closes the current LAMMPS log file, opens a new file with the specified name, and begins logging information to it. If the specified file name is *none*, then no new log file is opened. If the optional keyword *append* is specified, then output will be appended to an existing log file, instead of overwriting it.

If multiple processor partitions are being used, the file name should be a variable, so that different processors do not attempt to write to the same log file.

The file "log.lammps" is the default log file for a LAMMPS run. The name of the initial log file can also be set by the *-log command-line switch*.

1.50.4 Restrictions

none

1.50.5 Related commands

none

1.50.6 Default

The default LAMMPS log file is named log.lammps

1.51 mass command

1.51.1 Syntax

```
mass I value
```

- I = atom type (see asterisk form below), or type label
- value = mass

1.51.2 Examples

```
mass 1 1.0
mass * 62.5
mass 2* 62.5

labelmap atom 1 C
mass C 12.01
```

1.51.3 Description

Set the mass for all atoms of one or more atom types. Per-type mass values can also be set in the *read_data* data file using the "Masses" keyword. See the *units* command for what mass units to use.

The I index can be specified in one of several ways. An explicit numeric value can be used, as in the first example above. Or I can be a type label, which is an alphanumeric string defined by the *labelmap* command or in a section of a data file read by the *read_data* command, and which converts internally to a numeric type. Or a wild-card asterisk can be used to set the mass for multiple atom types. This takes the form "*" or "*n" or "n*" or "m*n", where m and n are numbers. If N = the number of atom types, then an asterisk with no numeric values means all types from 1 to N. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to N (inclusive). A middle asterisk means all types from m to n (inclusive).

A line in a *data file* that follows the "Masses" keyword specifies mass using the same format as the arguments of the mass command in an input script, except that no wild-card asterisk can be used. For example, under the "Masses" section of a data file, the line that corresponds to the first example above would be listed as

```
1 1.0
```

Note that the mass command can only be used if the *atom style* requires per-type atom mass to be set. Currently, all but the *sphere* and *ellipsoid* and *peri* styles do. They require mass to be set for individual particles, not types. Per-atom masses are defined in the data file read by the *read_data* command, or set to default values by the *create_atoms* command. Per-atom masses can also be set to new values by the *set mass* or *set density* commands.

Also note that *pair_style eam* and *pair_style bop* commands define the masses of atom types in their respective potential files, in which case the mass command is normally not used.

If you define a *hybrid atom style* which includes one (or more) sub-styles which require per-type mass and one (or more) sub-styles which require per-atom mass, then you must define both. However, in this case the per-type mass will be ignored; only the per-atom mass will be used by LAMMPS.

1.51.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

All masses must be defined before a simulation is run. They must also all be defined before a *velocity* or *fix shake* command is used.

The mass assigned to any type or atom must be > 0.0.

1.51.5 Related commands

none

1.51.6 Default

none

1.52 mdi command

1.52.1 Syntax

```
mdi option args
```

- option = *engine* or *plugin* or *connect* or *exit*

  engine args = zero or more keyword/args pairs
    keywords = elements
      elements args = N_1 N_2 ... N_ntypes
        N_1,N_2,...N_ntypes = chemical symbol for each of ntypes LAMMPS atom types
  plugin args = name keyword value keyword value ...
    name = name of plugin library (e.g., lammps means a liblammps.so library will be loaded)
    keyword/value pairs in any order, some are required, some are optional
    keywords = mdi or infile or extra or command
      mdi value = args passed to MDI for driver to operate with plugins (required)
      infile value = filename the engine will read at start-up (optional)
      extra value = aditional command-line args to pass to engine library when loaded (optional)
      command value = a LAMMPS input script command to execute (required)
  connect args = none
  exit args = none

1.52.2 Examples

```
mdi engine
mdi engine elements Al Cu
mdi plugin lammps mdi "-role ENGINE -name lammps -method LINK" &
        infile in.aimd.engine extra "-log log.aimd.engine.plugin" &
        command "run 5"
mdi connect
mdi exit
```

1.52.3 Description

This command implements operations within LAMMPS to use the *MDI Library <https://molssi-mdi.github.io/MDI_Library/html/index.html>* for coupling to other codes in a client/server protocol.

See the Howto MDI doc page for a discussion of all the different ways 2 or more codes can interact via MDI.

The examples/mdi directory has examples which use LAMMPS in 4 different modes: as a driver using an engine as either a stand-alone code or as a plugin, and as an engine operating as either a stand-alone code or as a plugin. The README file in that directory shows how to launch and couple codes for all the 4 usage modes, and so they communicate via the MDI library using either MPI or sockets.

The scripts in that directory illustrate the use of all the options for this command.

The *engine* option enables LAMMPS to act as an MDI engine (server), responding to requests from an MDI driver (client) code.

The *plugin* option enables LAMMPS to act as an MDI driver (client), and load the MDI engine (server) code as a library plugin. In this case the MDI engine is a library plugin. An MDI engine can also be a stand-alone code, launched separately from LAMMPS, in which case the mdi plugin command is not used.

The *connect* and *exit* options are only used when LAMMPS is acting as an MDI driver. As explained below, these options are normally not needed, except for a specific kind of use case.

The *mdi engine* command is used to make LAMMPS operate as an MDI engine. It is typically used in an input script after LAMMPS has setup the system it is going to model consistent with what the driver code expects. Depending on when the driver code tells the LAMMPS engine to exit, other commands can be executed after this command, but typically it is used at the end of a LAMMPS input script.

To act as an MDI engine operating as an MD code (or surrogate QM code), this is the list of standard MDI commands issued by a driver code which LAMMPS currently recognizes. Using standard commands defined by the MDI library means that a driver code can work interchangeably with LAMMPS or other MD codes or with QM codes which support the MDI standard. See more details about these commands in the MDI library documentation

These commands are valid at the @DEFAULT node defined by MDI. Commands that start with ">" mean the driver is sending information to LAMMPS. Commands that start with "<" are requests by the driver for LAMMPS to send it information. Commands that start with an alphabetic letter perform actions. Commands that start with "@" are MDI "node" commands, which are described further below.

Command name	Action
>CELL or <CELL	Send/request 3 simulation box edge vectors (9 values)
>CELL_DISPL or <CELL_DISPL	Send/request displacement of the simulation box from the origin (3 values)
>CHARGES or <CHARGES	Send/request charge on each atom (N values)
>COORDS or <COORDS	Send/request coordinates of each atom (3N values)
>ELEMENTS	Send elements (atomic numbers) for each atom (N values)
<ENERGY	Request total energy (potential + kinetic) of the system (1 value)
>FORCES or <FORCES	Send/request forces on each atom (3N values)
>+FORCES	Send forces to add to each atom (3N values)
<LABELS	Request string label of each atom (N values)
<MASSES	Request mass of each atom (N values)
MD	Perform an MD simulation for N timesteps (most recent >NSTEPS value)
OPTG	Perform an energy minimization to convergence (most recent >TOLERANCE values)
>NATOMS or <NATOMS	Sends/request number of atoms in the system (1 value)
>NSTEPS	Send number of timesteps for next MD dynamics run via MD command
<PE	Request potential energy of the system (1 value)
<STRESS	Request symmetric stress tensor (virial) of the system (9 values)
>TOLERANCE	Send 4 tolerance parameters for next MD minimization via OPTG command
>TYPES or <TYPES	Send/request the LAMMPS atom type for each atom (N values)
>VELOCITIES or <VELOCITIES	Send/request the velocity of each atom (3N values)
@INIT_MD or @INIT_OPTG	Driver tells LAMMPS to start single-step dynamics or minimization (see below)
EXIT	Driver tells LAMMPS to exit engine mode

Note

The <ENERGY, <FORCES, <PE, and <STRESS commands trigger LAMMPS to compute atomic interactions for the current configuration of atoms and size/shape of the simulation box. I.e. LAMMPS invokes its pair, bond, angle, ..., kspace styles. If the driver is updating the atom coordinates and/or box incrementally (as in an MD simulation which the driver is managing), then the LAMMPS engine will do the same, and only occasionally trigger neighbor list builds. If the change in atom positions is large (since the previous >COORDS command), then LAMMPS will do a more expensive operation to migrate atoms to new processors as needed and re-neighbor. If the >NATOMS or >TYPES or >ELEMENTS commands have been sent (since the previous >COORDS command), then LAMMPS assumes the system is new and re-initializes an entirely new simulation.

Note

The >TYPES or >ELEMENTS commands are how the MDI driver tells the LAMMPS engine which LAMMPS atom type to assign to each atom. If both the MDI driver and the LAMMPS engine are initialized so that atom type values are consistent in both codes, then the >TYPES command can be used. If not, the optional *elements* keyword can be used to specify what element each LAMMPS atom type corresponds to. This is specified by the chemical symbol of the element, e.g. C or Al or Si. A symbol must be specified for each of the ntypes LAMMPS atom types. Each LAMMPS type must map to a unique element; two or more types cannot map to the same element. Ntypes is typically specified via the *create_box* command or in the data file read by the *read_data* command. Once this has been done, the MDI driver can send an >ELEMENTS command to the LAMMPS driver with the atomic number of each atom and the LAMMPS engine will be able to map it to a LAMMPS atom type.

The MD and OPTG commands perform an entire MD simulation or energy minimization (to convergence) with no

communication from the driver until the simulation is complete. By contrast, the @INIT_MD and @INIT_OPTG commands allow the driver to communicate with the engine at each timestep of a dynamics run or iteration of a minimization; see more info below.

The MD command performs a simulation using the most recent >NSTEPS value. The OPTG command performs a minimization using the 4 convergence parameters from the most recent >TOLERANCE command. The 4 parameters sent are those used by the *minimize* command in LAMMPS: etol, ftol, maxiter, and maxeval.

The mdi engine command also implements the following custom MDI commands which are LAMMPS-specific. These commands are also valid at the @DEFAULT node defined by MDI:

- – Command name
  - – Action
- – >NBYTES
  - – Send # of datums in a subsequent command (1 value)
- – >COMMAND
  - – Send a LAMMPS input script command as a string (Nbytes in length)
- – >COMMANDS
  - – Send multiple LAMMPS input script commands as a newline-separated string (Nbytes in length)
- – >INFILE
  - – Send filename of an input script to execute (filename Nbytes in length)
- – <KE
  - – Request kinetic energy of the system (1 value)

Note that other custom commands can easily be added if these are not sufficient to support what a user-written driver code needs. Code to support new commands can be added to the MDI package within LAMMPS, specifically to the src/MDI/mdi_engine.cpp file.

MDI also defines a standard mechanism for the driver to request that an MD engine (LAMMPS) perform a dynamics simulation one step at a time or an energy minimization one iteration at a time. This is so that the driver can (optionally) communicate with LAMMPS at intermediate points of the timestep or iteration by issuing MDI node commands which start with "@".

To tell LAMMPS to run dynamics in single-step mode, the driver sends as @INIT_MD command followed by the these commands. The driver can interact with LAMMPS at 3 node locations within each timestep: @COORDS, @FORCES, @ENDSTEP:

- – Command name
  - – Action
- – @COORDS
  - – Proceed to next @COORDS node = post-integrate location in LAMMPS timestep
- – @FORCES
  - – Proceed to next @FORCES node = post-force location in LAMMPS timestep
- – @ENDSTEP
  - – Proceed to next @ENDSTEP node = end-of-step location in LAMMPS timestep
- – @DEFAULT
  - – Exit MD simulation, return to @DEFAULT node

- – EXIT
  - – Driver tells LAMMPS to exit the MD simulation and engine mode

To tell LAMMPS to run an energy minimization in single-iteration mode. The driver can interact with LAMMPS at 2 node locations within each iteration of the minimizer: @COORDS, @FORCES:

- – Command name
  - – Action
- – @COORDS
  - – Proceed to next @COORDS node = min-pre-force location in LAMMPS min iteration
- – @FORCES
  - – Proceed to next @FORCES node = min-post-force location in LAMMPS min iteration
- – @DEFAULT
  - – Exit minimization, return to @DEFAULT node
- – EXIT
  - – Driver tells LAMMPS to exit the minimization and engine mode

While LAMMPS is at its @COORDS node, the following standard MDI commands are supported, as documented above: >COORDS or <COORDS, @COORDS, @FORCES, @ENDSTEP, @DEFAULT, EXIT.

While LAMMPS is at its @FORCES node, the following standard MDI commands are supported, as documented above: <COORDS, <ENERGY, >FORCES or >+FORCES or <FORCES, <KE, <PE, <STRESS, @COORDS, @FORCES, @ENDSTEP, @DEFAULT, EXIT.

While LAMMPS is at its @ENDSTEP node, the following standard MDI commands are supported, as documented above: <ENERGY, <FORCES, <KE, <PE, <STRESS, @COORDS, @FORCES, @ENDSTEP, @DEFAULT, EXIT.

The *mdi plugin* command is used to make LAMMPS operate as an MDI driver which loads an MDI engine as a plugin library. It is typically used in an input script after LAMMPS has setup the system it is going to model consistent with the engine code.

The *name* argument specifies which plugin library to load. A name like "lammps" is converted to a filename liblammps.so. The path for where this file is located is specified by the -plugin_path switch within the -mdi command-line switch, which is specified when LAMMPS is launched. See the examples/mdi/README files for examples of how this is done.

The *mdi* keyword is required and is used as the -mdi argument passed to the library when it is launched. The -role and -method settings are required. The -name setting can be anything you choose. MDI drivers and engines can query their names to verify they are values they expect.

The *infile* keyword is optional. It sets the name of an input script which the engine will open and process. MDI will pass it as a command-line argument to the library when it is launched. The file typically contains settings that an MD or QM code will use for its calculations.

The *extra* keyword is optional. It contains additional command-line arguments which MDI will pass to the library when it is launched.

The *command* keyword is required. It specifies a LAMMPS input script command (as a single argument in quotes if it is multiple words). Once the plugin library is launched, LAMMPS will execute this command. Other previously-defined commands in the input script, such as the *fix mdi/qm* command, should perform MDI communication with the engine, while the specified *command* executes. Note that if *command* is an *include* command, then it could specify a filename with multiple LAMMPS commands.

Note

When the *command* is complete, LAMMPS will send an MDI EXIT command to the plugin engine and the plugin will be removed. The "mdi plugin" command will then exit and the next command (if any) in the LAMMPS input script will be processed. A subsequent "mdi plugin" command could then load the same or a different MDI plugin if desired.

The *mdi connect* and *mdi exit* commands are only used when LAMMPS is operating as an MDI driver. And when other LAMMPS command(s) which send MDI commands and associated data to/from the MDI engine are not able to initiate and terminate the connection to the engine code.

The only current MDI driver command in LAMMPS is the *fix mdi/qm* command. If it is only used once in an input script then it can initiate and terminate the connection, but if it is being issued multiple times (e.g., in a loop that issues a *clear* command), then it cannot initiate or terminate the connection multiple times. Instead, the *mdi connect* and *mdi exit* commands should be used outside the loop to initiate or terminate the connection.

See the examples/mdi/in.series.driver script for an example of how this is done. The LOOP in that script is reading a series of data file configurations and passing them to an MDI engine (e.g., quantum code) for energy and force evaluation. A *clear* command inside the loop wipes out the current system so a new one can be defined. This operation also destroys all fixes. So the *fix mdi/qm* command is issued once per loop iteration. Note that it includes a "connect no" option which disables the initiate/terminate logic within that fix.

1.52.4 Restrictions

This command is part of the MDI package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

To use LAMMPS in conjunction with other MDI-enabled atomistic codes, the *units* command should be used to specify *real* or *metal* units. This will ensure the correct unit conversions between LAMMPS and MDI units, which the other codes will also perform in their preferred units.

LAMMPS can also be used as an MDI engine in other unit choices it supports (e.g., *lj*), but then no unit conversion is performed.

1.52.5 Related commands

fix mdi/qm

1.52.6 Default

None

1.53 min_modify command

1.53.1 Syntax

```
min_modify keyword values ...
```

- one or more keyword/value pairs may be listed

keyword = dmax or line or norm or alpha_damp or discrete_factor or integrator or abcfire or tmax
  dmax value = max
    max = maximum distance for line search to move (distance units)
  line value = backtrack or quadratic or forcezero or spin_cubic or spin_none
    backtrack,quadratic,forcezero,spin_cubic,spin_none = style of linesearch to use
  norm value = two or inf or max
    two = Euclidean two-norm (length of 3N vector)
    inf = max force component across all 3-vectors
    max = max force norm across all 3-vectors
  alpha_damp value = damping
    damping = fictitious magnetic damping for spin minimization (adim)
  discrete_factor value = factor
    factor = discretization factor for adaptive spin timestep (adim)
  integrator value = eulerimplicit or verlet or leapfrog or eulerexplicit
    time integration scheme for fire minimization
  abcfire value = yes or no (default no)
    yes = use ABC-FIRE variant of fire minimization style
    no = use default FIRE variant of fire minimization style
  tmax value = factor
    factor = maximum adaptive timestep for fire minimization (adim)

1.53.2 Examples

```
min_modify dmax 0.2
min_modify integrator verlet tmax 4
```

1.53.3 Description

This command sets parameters that affect the energy minimization algorithms selected by the *min_style* command. The various settings may affect the convergence rate and overall number of force evaluations required by a minimization, so users can experiment with these parameters to tune their minimizations.

The *cg* and *sd* minimization styles have an outer iteration and an inner iteration which is steps along a one-dimensional line search in a particular search direction. The *dmax* parameter is how far any atom can move in a single line search in any dimension (x, y, or z). For the *quickmin* and *fire* minimization styles, the *dmax* setting is how far any atom can move in a single iteration (timestep). Thus a value of 0.1 in real *units* means no atom will move further than 0.1 Angstroms in a single outer iteration. This prevents highly overlapped atoms from being moved long distances (e.g. through another atom) due to large forces.

The choice of line search algorithm for the *cg* and *sd* minimization styles can be selected via the *line* keyword. The default *quadratic* line search algorithm starts out using the robust backtracking method described below. However, once the system gets close to a local minimum and the linesearch steps get small, so that the energy is approximately quadratic in the step length, it uses the estimated location of zero gradient as the linesearch step, provided the energy

change is downhill. This becomes more efficient than backtracking for highly-converged relaxations. The *forcezero* line search algorithm is similar to *quadratic*. It may be more efficient than *quadratic* on some systems.

The backtracking search is robust and should always find a local energy minimum. However, it will "converge" when it can no longer reduce the energy of the system. Individual atom forces may still be larger than desired at this point, because the energy change is measured as the difference of two large values (energy before and energy after) and that difference may be smaller than machine epsilon even if atoms could move in the gradient direction to reduce forces further.

The choice of a norm can be modified for the min styles *cg*, *sd*, *quickmin*, *fire*, *fire/old*, *spin*, *spin/cg* and *spin/lbfgs* using the *norm* keyword. The default *two* norm computes the 2-norm (Euclidean length) of the global force vector:

$$||\vec{F}||_2 = \sqrt{\vec{F}_1^2 + \cdots + \vec{F}_N^2}$$

The *max* norm computes the length of the 3-vector force for each atom (2-norm), and takes the maximum value of those across all atoms

$$||\vec{F}||_{max} = \max\left(||\vec{F}_1||, \cdots, ||\vec{F}_N||\right)$$

The *inf* norm takes the maximum component across the forces of all atoms in the system:

$$||\vec{F}||_{inf} = \max\left(|F_1^1|, |F_1^2|, |F_1^3| \cdots, |F_N^1|, |F_N^2|, |F_N^3|\right)$$

For the min styles *spin*, *spin/cg* and *spin/lbfgs*, the force norm is replaced by the spin-torque norm.

Keywords *alpha_damp* and *discrete_factor* only make sense when a *min_spin* command is declared. Keyword *alpha_damp* defines an analog of a magnetic damping. It defines a relaxation rate toward an equilibrium for a given magnetic system. Keyword *discrete_factor* defines a discretization factor for the adaptive timestep used in the *spin* minimization. See *min_spin* for more information about those quantities.

The choice of a line search algorithm for the *spin/cg* and *spin/lbfgs* styles can be specified via the *line* keyword. The *spin_cubic* and *spin_none* keywords only make sense when one of those two minimization styles is declared. The *spin_cubic* performs the line search based on a cubic interpolation of the energy along the search direction. The *spin_none* keyword deactivates the line search procedure. The *spin_none* is a default value for *line* keyword for both *spin/lbfgs* and *spin/cg*. Convergence of *spin/lbfgs* can be more robust if *spin_cubic* line search is used.

The Newton *integrator* used for *fire* minimization can be selected to be either the symplectic Euler (*eulerimplicit*), velocity Verlet (*verlet*), Leapfrog (*leapfrog*) or non-symplectic forward Euler (*eulerexplicit*). The keyword *tmax* defines the maximum value for the adaptive timestep during a *fire* minimization. It is a multiplication factor applied to the current *timestep* (not in time unit). For example, *tmax* = 4.0 with a *timestep* of 2fs, means that the maximum value the timestep can reach during a *fire* minimization is 4fs. Note that parameter defaults has been chosen to be reliable in most cases, but one should consider adjusting *timestep* and *tmax* to optimize the minimization for large or complex systems. Other parameters of the *fire* minimization can be tuned (*tmin*, *delaystep*, *dtgrow*, *dtshrink*, *alpha0*, and *alphashrink*). Please refer to the references describing the *min_style fire*. An additional stopping criteria *vdfmax* is used by *fire* in order to avoid unnecessary looping when it is reasonable to think the system will not be relaxed further. Note that in this case the system will NOT have reached your minimization criteria. This could happen when the system comes to be stuck in a local basin of the phase space. *vdfmax* is the maximum number of consecutive iterations with P(t) < 0.

Added in version 8Feb2023.

The *abcfire* keyword allows to activate the ABC-FIRE variant of the *fire* minimization algorithm. ABC-FIRE introduces an additional factor that modifies the bias and scaling of the velocities of the atoms during the mixing step *(Echeverri Restrepo)*. This can lead to faster convergence of the minimizer.

The *min_style fire* is an optimized implementation of *min_style fire/old*. It can however behave similarly to the *fire/old* style by using the following set of parameters:

```
min_modify integrator eulerexplicit tmax 10.0 tmin 0.0 delaystep 5 &
        dtgrow 1.1 dtshrink 0.5 alpha0 0.1 alphashrink 0.99 &
        vdfmax 100000 halfstepback no initialdelay no
```

1.53.4 Restrictions

For magnetic GNEB calculations, only *spin_none* value for *line* keyword can be used when minimization styles *spin/cg* and *spin/lbfgs* are employed. See *neb/spin* for more explanation.

1.53.5 Related commands

min_style, *minimize*

1.53.6 Default

The option defaults are dmax = 0.1, line = quadratic and norm = two.

For the *spin*, *spin/cg* and *spin/lbfgs* styles, the option defaults are alpha_damp = 1.0, discrete_factor = 10.0, line = spin_none, and norm = euclidean.

For the *fire* style, the option defaults are integrator = eulerimplicit, tmax = 10.0, tmin = 0.02, delaystep = 20, dtgrow = 1.1, dtshrink = 0.5, alpha0 = 0.25, alphashrink = 0.99, vdfmax = 2000, halfstepback = yes and initialdelay = yes.

(EcheverriRestrepo) Echeverri Restrepo, Andric, Comput Mater Sci, 218, 111978 (2023).

1.54 min_style spin command

1.55 min_style spin/cg command

1.56 min_style spin/lbfgs command

1.56.1 Syntax

```
min_style spin
min_style spin/cg
min_style spin/lbfgs
```

1.56.2 Examples

```
min_style  spin/lbfgs
min_modify line spin_cubic discrete_factor 10.0
```

1.56.3 Description

Apply a minimization algorithm to use when a *minimize* command is performed.

Style *spin* defines a damped spin dynamics with an adaptive timestep, according to:

$$\frac{d\vec{s}_i}{dt} = \lambda\, \vec{s}_i \times \left(\vec{\omega}_i \times \vec{s}_i\right)$$

with λ a damping coefficient (similar to a magnetic damping). λ can be defined by setting the *alpha_damp* keyword with the *min_modify* command.

The minimization procedure solves this equation using an adaptive timestep. The value of this timestep is defined by the largest precession frequency that has to be solved in the system:

$$\Delta t_{\max} = \frac{2\pi}{\kappa\, |\vec{\omega}_{\max}|}$$

with $|\vec{\omega}_{\max}|$ the norm of the largest precession frequency in the system (across all processes, and across all replicas if a spin/neb calculation is performed).

κ defines a discretization factor *discrete_factor* for the definition of this timestep. *discrete_factor* can be defined with the *min_modify* command.

Style *spin/cg* defines an orthogonal spin optimization (OSO) combined to a conjugate gradient (CG) algorithm. The *min_modify* command can be used to couple the *spin/cg* to a line search procedure, and to modify the discretization factor *discrete_factor*. By default, style *spin/cg* does not employ the line search procedure and uses the adaptive timestep technique in the same way as style *spin*.

Style *spin/lbfgs* defines an orthogonal spin optimization (OSO) combined to a limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm. By default, style *spin/lbfgs* does not employ line search procedure. If the line search procedure is not used then the discrete factor defines the maximum root mean squared rotation angle of spins by equation *pi/(5*Kappa)*. The default value for Kappa is 10. The *spin_cubic* line search option can improve the convergence of the *spin/lbfgs* algorithm.

The *min_modify* command can be used to activate the line search procedure, and to modify the discretization factor *discrete_factor*.

For more information about styles *spin/cg* and *spin/lbfgs*, see their implementation reported in *(Ivanov)*.

Note

All the *spin* styles replace the force tolerance by a torque tolerance. See *minimize* for more explanation.

Note

The *spin/cg* and *spin/lbfgs* styles can be used for magnetic NEB calculations only if the line search procedure is deactivated. See *neb/spin* for more explanation.

1.56.4 Restrictions

The *spin*, *spin/cg*, and *spin/lbfgps* styles are part of the SPIN package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This minimization procedure is only applied to spin degrees of freedom for a frozen lattice configuration.

1.56.5 Related commands

min_style, *minimize*, *min_modify*

1.56.6 Default

The option defaults are *alpha_damp* = 1.0, *discrete_factor* = 10.0, *line* = spin_none and *norm* = euclidean.

(Ivanov) Ivanov, Uzdin, Jonsson. arXiv preprint arXiv:1904.02669, (2019).

1.57 min_style cg command

1.58 min_style hftn command

1.59 min_style sd command

1.60 min_style quickmin command

1.61 min_style fire command

1.62 min_style spin command

1.63 min_style spin/cg command

1.64 min_style spin/lbfgs command

1.64.1 Syntax

```
min_style style
```

- style = *cg* or *hftn* or *sd* or *quickmin* or *fire* or *spin* or *spin/cg* or *spin/lbfgs*

  spin is discussed briefly here and fully on min_style spin doc page
  spin/cg is discussed briefly here and fully on min_style spin doc page
  spin/lbfgs is discussed briefly here and fully on min_style spin doc page

1.64.2 Examples

```
min_style cg
min_style fire
min_style spin
```

1.64.3 Description

Choose a minimization algorithm to use when a *minimize* command is performed.

Style *cg* is the Polak-Ribiere version of the conjugate gradient (CG) algorithm. At each iteration the force gradient is combined with the previous iteration information to compute a new search direction perpendicular (conjugate) to the previous search direction. The PR variant affects how the direction is chosen and how the CG method is restarted when it ceases to make progress. The PR variant is thought to be the most effective CG choice for most problems.

Style *hftn* is a Hessian-free truncated Newton algorithm. At each iteration a quadratic model of the energy potential is solved by a conjugate gradient inner iteration. The Hessian (second derivatives) of the energy is not formed directly, but approximated in each conjugate search direction by a finite difference directional derivative. When close to an energy minimum, the algorithm behaves like a Newton method and exhibits a quadratic convergence rate to high accuracy. In most cases the behavior of *hftn* is similar to *cg*, but it offers an alternative if *cg* seems to perform poorly. This style is not affected by the *min_modify* command.

Style *sd* is a steepest descent algorithm. At each iteration, the search direction is set to the downhill direction corresponding to the force vector (negative gradient of energy). Typically, steepest descent will not converge as quickly as CG, but may be more robust in some situations.

Style *quickmin* is a damped dynamics method described in *(Sheppard)*, where the damping parameter is related to the projection of the velocity vector along the current force vector for each atom. The velocity of each atom is initialized to 0.0 by this style, at the beginning of a minimization.

Style *fire* is a damped dynamics method described in *(Bitzek)*, which is similar to *quickmin* but adds a variable timestep and alters the projection operation to maintain components of the velocity non-parallel to the current force vector. The velocity of each atom is initialized to 0.0 by this style, at the beginning of a minimization. This style correspond to an optimized version described in *(Guenole)* that include different time integration schemes and default parameters. The default parameters can be modified with the command *min_modify*.

Style *spin* is a damped spin dynamics with an adaptive timestep.

Style *spin/cg* uses an orthogonal spin optimization (OSO) combined to a conjugate gradient (CG) approach to minimize spin configurations.

Style *spin/lbfgs* uses an orthogonal spin optimization (OSO) combined to a limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) approach to minimize spin configurations.

See the *min/spin* page for more information about the *spin*, *spin/cg* and *spin/lbfgs* styles.

Either the *quickmin* or the *fire* styles are useful in the context of nudged elastic band (NEB) calculations via the *neb* command.

Either the *spin*, *spin/cg*, or *spin/lbfgs* styles are useful in the context of magnetic geodesic nudged elastic band (GNEB) calculations via the *neb/spin* command.

Note

The damped dynamic minimizers use whatever timestep you have defined via the *timestep* command. Often they will converge more quickly if you use a timestep about 10x larger than you would normally use for dynamics

simulations. For *fire*, the default timestep is recommended to be equal to the one you would normally use for dynamics simulations.

Note

The *quickmin*, *fire*, *hftn*, and *cg/kk* styles do not yet support the use of the *fix box/relax* command or minimizations involving the electron radius in *eFF* models.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

1.64.4 Restrictions

The *spin*, *spin/cg*, and *spin/lbfgps* styles are part of the SPIN package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

1.64.5 Related commands

min_modify, *minimize*, *neb*

1.64.6 Default

```
min_style cg
```

(Sheppard) Sheppard, Terrell, Henkelman, J Chem Phys, 128, 134106 (2008). See ref 1 in this paper for original reference to Qmin in Jonsson, Mills, Jacobsen.

(Bitzek) Bitzek, Koskinen, Gahler, Moseler, Gumbsch, Phys Rev Lett, 97, 170201 (2006).

(Guenole) Guenole, Noehring, Vaid, Houlle, Xie, Prakash, Bitzek, Comput Mater Sci, 175, 109584 (2020).

1.65 minimize command

Accelerator Variant: minimize/kk

1.65.1 Syntax

```
minimize etol ftol maxiter maxeval
```

- etol = stopping tolerance for energy (unitless)
- ftol = stopping tolerance for force (force units)
- maxiter = max iterations of minimizer
- maxeval = max number of force/energy evaluations

1.65.2 Examples

```
minimize 1.0e-4 1.0e-6 100 1000
minimize 0.0 1.0e-8 1000 100000
```

1.65.3 Description

Perform an energy minimization of the system, by iteratively adjusting atom coordinates. Iterations are terminated when one of the stopping criteria is satisfied. At that point the configuration will hopefully be in a local potential energy minimum. More precisely, the configuration should approximate a critical point for the objective function (see below), which may or may not be a local minimum.

The minimization algorithm used is set by the *min_style* command. Other options are set by the *min_modify* command. Minimize commands can be interspersed with *run* commands to alternate between relaxation and dynamics. The minimizers bound the distance atoms may move in one iteration, so that you can relax systems with highly overlapped atoms (large energies and forces) by pushing the atoms off of each other.

Neighbor list update settings

The distance that atoms can move during individual minimization steps can be quite large, especially at the beginning of a minimization. Thus *neighbor list settings* of *every* = *1* and *delay* = *0* are **required**. This may be combined with either *check* = *no* (always update the neighbor list) or *check* = *yes* (only update the neighbor list if at least one atom has moved more than half the *neighbor list skin* distance since the last reneighboring). Using *check* = *yes* is recommended since it avoids unneeded reneighboring steps when the system is closer to the minimum and thus atoms move only small distances. Using *check* = *no* may be required for debugging or when coupling LAMMPS with external codes that require a predictable sequence of neighbor list updates.

If the settings are **not** *every* = *1* and *delay* = *0*, LAMMPS will temporarily apply a *neigh_modify every 1 delay 0 check yes* setting during the minimization and restore the original setting at the end of the minimization. A corresponding message will be printed to the screen and log file, if this happens.

Alternate means of relaxing a system are to run dynamics with a small or *limited timestep*. Or dynamics can be run using *fix viscous* to impose a damping force that slowly drains all kinetic energy from the system. The *pair_style soft* potential can be used to un-overlap atoms while running dynamics.

Note that you can minimize some atoms in the system while holding the coordinates of other atoms fixed by applying *fix setforce 0.0 0.0 0.0* to the other atoms. See a more detailed discussion of *using fixes while minimizing below*.

The *minimization styles* *cg*, *sd*, and *hftn* involves an outer iteration loop which sets the search direction along which atom coordinates are changed. An inner iteration is then performed using a line search algorithm. The line search typically evaluates forces and energies several times to set new coordinates. Currently, a backtracking algorithm is used which may not be optimal in terms of the number of force evaluations performed, but appears to be more robust than previous line searches we have tried. The backtracking method is described in Nocedal and Wright's Numerical Optimization (Procedure 3.1 on p 41).

The *minimization styles* *quickmin*, *fire* and *fire/old* perform damped dynamics using an Euler integration step. Thus they require a *timestep* be defined.

Note

The damped dynamic minimizer algorithms will use the timestep you have defined via the *timestep* command or its default value. Often they will converge more quickly if you use a timestep about 10x larger than you would normally use for regular molecular dynamics simulations.

In all cases, the objective function being minimized is the total potential energy of the system as a function of the N atom coordinates:

$$E(r_1, r_2, \ldots, r_N) = \sum_{i,j} E_{pair}(r_i, r_j) + \sum_{ij} E_{bond}(r_i, r_j) + \sum_{ijk} E_{angle}(r_i, r_j, r_k) + \sum_{ijkl} E_{dihedral}(r_i, r_j, r_k, r_l) + \sum_{ijkl} E_{improper}(r_i, r_j, r_k, r_l) + \sum_{i} E_{fix}(r_i)$$

where the first term is the sum of all non-bonded *pairwise interactions* including *long-range Coulombic interactions*, the second through fifth terms are *bond*, *angle*, *dihedral*, and *improper* interactions respectively, and the last term is energy due to *fixes* which can act as constraints or apply force to atoms, such as through interaction with a wall. See the discussion below about how fix commands affect minimization.

The starting point for the minimization is the current configuration of the atoms.

The minimization procedure stops if any of several criteria are met:

- the change in energy between outer iterations is less than *etol*
- the 2-norm (length) of the global force vector is less than the *ftol*
- the line search fails because the step distance backtracks to 0.0
- the number of outer iterations or timesteps exceeds *maxiter*
- the number of total force evaluations exceeds *maxeval*

Note

the *minimization style* *spin*, *spin/cg*, and *spin/lbfgs* replace the force tolerance *ftol* by a torque tolerance. The minimization procedure stops if the 2-norm (length) of the torque vector on atom (defined as the cross product between the atomic spin and its precession vectors omega) is less than *ftol*, or if any of the other criteria are met. Torque have the same units as the energy.

Note

You can also use the *fix halt* command to specify a general criterion for exiting a minimization, that is a calculation performed on the state of the current system, as defined by an *equal-style variable*.

For the first criterion, the specified energy tolerance *etol* is unitless; it is met when the energy change between successive iterations divided by the energy magnitude is less than or equal to the tolerance. For example, a setting of 1.0e-4 for *etol* means an energy tolerance of one part in 10^4. For the damped dynamics minimizers this check is not performed for a few steps after velocities are reset to 0, otherwise the minimizer would prematurely converge.

For the second criterion, the specified force tolerance *ftol* is in force units, since it is the length of the global force vector for all atoms, e.g. a vector of size 3N for N atoms. Since many of the components will be near zero after minimization, you can think of *ftol* as an upper bound on the final force on any component of any atom. For example, a setting of 1.0e-4 for *ftol* means no x, y, or z component of force on any atom will be larger than 1.0e-4 (in force units) after minimization.

Either or both of the *etol* and *ftol* values can be set to 0.0, in which case some other criterion will terminate the minimization.

During a minimization, the outer iteration count is treated as a timestep. Output is triggered by this timestep, e.g. thermodynamic output or dump and restart files.

Using the *thermo_style custom* command with the *fmax* or *fnorm* keywords can be useful for monitoring the progress of the minimization. Note that these outputs will be calculated only from forces on the atoms, and will not include any extra degrees of freedom, such as from the *fix box/relax* command.

Following minimization, a statistical summary is printed that lists which convergence criterion caused the minimizer to stop, as well as information about the energy, force, final line search, and iteration counts. An example is as follows:

```
Minimization stats:
  Stopping criterion = max iterations
  Energy initial, next-to-last, final =
      -0.626828169302     -2.82642039062     -2.82643549739
  Force two-norm initial, final = 2052.1 91.9642
  Force max component initial, final = 346.048 9.78056
  Final line search alpha, max atom move = 2.23899e-06 2.18986e-05
  Iterations, force evaluations = 2000 12724
```

The 3 energy values are for before and after the minimization and on the next-to-last iteration. This is what the *etol* parameter checks.

The two-norm force values are the length of the global force vector before and after minimization. This is what the *ftol* parameter checks.

The max-component force values are the absolute value of the largest component (x,y,z) in the global force vector, i.e. the infinity-norm of the force vector.

The alpha parameter for the line-search, when multiplied by the max force component (on the last iteration), gives the max distance any atom moved during the last iteration. Alpha will be 0.0 if the line search could not reduce the energy. Even if alpha is non-zero, if the "max atom move" distance is tiny compared to typical atom coordinates, then it is possible the last iteration effectively caused no atom movement and thus the evaluated energy did not change and the minimizer terminated. Said another way, even with non-zero forces, it's possible the effect of those forces is to move atoms a distance less than machine precision, so that the energy cannot be further reduced.

The iterations and force evaluation values are what is checked by the *maxiter* and *maxeval* parameters.

Note

There are several force fields in LAMMPS which have discontinuities or other approximations which may prevent you from performing an energy minimization to tight tolerances. For example, you should use a *pair style* that goes to 0.0 at the cutoff distance when performing minimization (even if you later change it when running dynamics). If you do not do this, the total energy of the system will have discontinuities when the relative distance between any pair of atoms changes from cutoff *plus* epsilon to cutoff *minus* epsilon and the minimizer may thus behave poorly. Some of the many-body potentials use splines and other internal cutoffs that inherently have this problem. The *long-range Coulombic styles* (PPPM, Ewald) are approximate to within the user-specified tolerance, which means their energy and forces may not agree to a higher precision than the Kspace-specified tolerance. This agreement is further reduced when using tabulation to speed up the computation of the real-space part of the Coulomb interactions, which is enabled by default. In all these cases, the minimizer may give up and stop before finding a minimum to the specified energy or force tolerance.

Note that a cutoff Lennard-Jones potential (and others) can be shifted so that its energy is 0.0 at the cutoff via the *pair_modify* command. See the doc pages for individual *pair styles* for details. Note that most Coulombic potentials have a cutoff, unless versions with a long-range component are used (e.g. *pair_style lj/cut/coul/long*) or some other damping/smoothing schemes are used. The CHARMM potentials go to 0.0 at the cutoff (e.g. *pair_style lj/charmm/coul/charmm*), as do the GROMACS potentials (e.g. *pair_style lj/gromacs*).

If a soft potential (*pair_style soft*) is used the Astop value is used for the prefactor (no time dependence).

The *fix box/relax* command can be used to apply an external pressure to the simulation box and allow it to shrink/expand during the minimization.

Only a few other fixes (typically those that add forces) are invoked during minimization. See the doc pages for individual *fix* commands to see which ones are relevant. Current examples of fixes that can be used include:

- *fix addforce*
- *fix addtorque*
- *fix efield*
- *fix enforce2d*
- *fix indent*
- *fix lineforce*
- *fix planeforce*
- *fix setforce*
- *fix spring*
- *fix spring/self*
- *fix viscous*
- *fix wall*
- *fix wall/region*

Note

Some fixes which are invoked during minimization have an associated potential energy. For that energy to be included in the total potential energy of the system (the quantity being minimized), you MUST enable the *fix_modify energy* option for that fix. The doc pages for individual *fix* commands specify if this should be done.

Note

The minimizers in LAMMPS do not allow for bonds (or angles, etc) to be held fixed while atom coordinates are being relaxed, e.g. via *fix shake* or *fix rigid*. See more info in the Restrictions section below.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

1.65.4 Restrictions

Features that are not yet implemented are listed here, in case someone knows how they could be coded:

It is an error to use *fix shake* with minimization because it turns off bonds that should be included in the potential energy of the system. The effect of a fix shake can be approximated during a minimization by using stiff spring constants for the bonds and/or angles that would normally be constrained by the SHAKE algorithm.

Fix rigid is also not supported by minimization. It is not an error to have it defined, but the energy minimization will not keep the defined body(s) rigid during the minimization. Note that if bonds, angles, etc internal to a rigid body have been turned off (e.g. via *neigh_modify exclude*), they will not contribute to the potential energy which is probably not what is desired.

Pair potentials that produce torque on a particle (e.g. *granular potentials* or the *GayBerne potential* for ellipsoidal particles) are not relaxed by a minimization. More specifically, radial relaxations are induced, but no rotations are induced by a minimization, so such a system will not fully relax.

1.65.5 Related commands

min_modify, *min_style*, *run_style*

1.65.6 Default

none

1.66 molecule command

1.66.1 Syntax

```
molecule ID file1 keyword values ... file2 keyword values ... fileN ...
```

- ID = user-assigned name for the molecule template
- file1,file2,... = names of files containing molecule descriptions
- zero or more keyword/value pairs may be appended after each file
- keyword = *offset* or *toff* or *boff* or *aoff* or *doff* or *ioff* or *scale*

```
offset values = Toff Boff Aoff Doff Ioff
  Toff = offset to add to atom types
  Boff = offset to add to bond types
  Aoff = offset to add to angle types
  Doff = offset to add to dihedral types
  Ioff = offset to add to improper types
toff value = Toff
  Toff = offset to add to atom types
boff value = Boff
  Boff = offset to add to bond types
aoff value = Aoff
  Aoff = offset to add to angle types
doff value = Doff
  Doff = offset to add to dihedral types
ioff value = Ioff
  Ioff = offset to add to improper types
scale value = sfactor
  sfactor = scale factor to apply to the size and mass of the molecule
```

1.66.2 Examples

```
molecule 1 mymol.txt
molecule 1 co2.txt h2o.txt
molecule CO2 co2.txt boff 3 aoff 2
molecule 1 mymol.txt offset 6 9 18 23 14
molecule objects file.1 scale 1.5 file.1 scale 2.0 file.2 scale 1.3
```

1.66.3 Description

Define a molecule template that can be used as part of other LAMMPS commands, typically to define a collection of particles as a bonded molecule or a rigid body. Commands that currently use molecule templates include:

- *fix deposit*
- *fix pour*
- *fix rigid/small*
- *fix shake*
- *fix gcmc*

- *fix bond/react*
- *create_atoms*
- *atom_style template*

The ID of a molecule template can only contain alphanumeric characters and underscores.

A single template can contain multiple molecules, listed one per file. Some of the commands listed above currently use only the first molecule in the template, and will issue a warning if the template contains multiple molecules. The *atom_style template* command allows multiple-molecule templates to define a system with more than one templated molecule.

Each filename can be followed by optional keywords which are applied only to the molecule in the file as used in this template. This is to make it easy to use the same molecule file in different molecule templates or in different simulations. You can specify the same file multiple times with different optional keywords.

The *offset*, *toff*, *boff*, *aoff*, *doff*, *ioff* keywords add the specified offset values to the atom types, bond types, angle types, dihedral types, and/or improper types as they are read from the molecule file. E.g. if *toff* = 2, and the file uses atom types 1,2,3, then each created molecule will have atom types 3,4,5. For the *offset* keyword, all five offset values must be specified, but individual values will be ignored if the molecule template does not use that attribute (e.g. no bonds).

Note

Offsets are **ignored** on lines using type labels, as the type labels will determine the actual types directly depending on the current *labelmap* settings.

The *scale* keyword scales the size of the molecule. This can be useful for modeling polydisperse granular rigid bodies. The scale factor is applied to each of these properties in the molecule file, if they are defined: the individual particle coordinates (Coords section), the individual mass of each particle (Masses section), the individual diameters of each particle (Diameters section), the total mass of the molecule (header keyword = mass), the center-of-mass of the molecule (header keyword = com), and the moments of inertia of the molecule (header keyword = inertia).

Note

The molecule command can be used to define molecules with bonds, angles, dihedrals, impropers, or special bond lists of neighbors within a molecular topology, so that you can later add the molecules to your simulation, via one or more of the commands listed above. Since this topology-related information requires that suitable storage is reserved when LAMMPS creates the simulation box (e.g. when using the *create_box* command or the *read_data* command) suitable space has to be reserved so you do not overflow those pre-allocated data structures when adding molecules later. Both the *create_box* command and the *read_data* command have "extra" options which ensure space is allocated for storing topology info for molecules that are added later.

1.66.4 Format of a molecule file

The format of an individual molecule file looks similar but is different than that of a data file read by the *read_data* commands. Here is a simple example for a TIP3P water molecule:

```
# Water molecule. TIP3P geometry
# header section:
3 atoms
2 bonds
1 angles

# body section:
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

1   1      1      2
2   1      1      3

Angles

1   1      2      1      3
```

A molecule file has a header and a body. The header appears first. The first line of the header and thus of the molecule file is *always* skipped; it typically contains a description of the file or a comment from the software that created the file.

Then lines are read one line at a time. Lines can have a trailing comment starting with '#' that is ignored. There *must* be at least one blank between any valid content and the comment. If the line is blank (i.e. contains only white-space after comments are deleted), it is skipped. If the line contains a header keyword, the corresponding value(s) is/are read from the line. A line that is *not* blank and does *not* contains a header keyword begins the body of the file.

The body of the file contains zero or more sections. The first line of a section has only a keyword. The next line is skipped. The remaining lines of the section contain values. The number of lines depends on the section keyword as described below. Zero or more blank lines can be used between sections. Sections can appear in any order, with a few exceptions as noted below.

These are the recognized header keywords. Header lines can come in any order. The numeric value(s) are read from the beginning of the line. The keyword should appear at the end of the line. All these settings have default values, as explained below. A line need only appear if the value(s) are different than the default, except when defining a *body* particle, which requires setting the number of *atoms* to 1, and setting the *inertia* in a specific section (see below).

Number(s)	Keyword	Meaning	Default Value
N	atoms	# of atoms N in molecule	0
Nb	bonds	# of bonds Nb in molecule	0
Na	angles	# of angles Na in molecule	0
Nd	dihedrals	# of dihedrals Nd in molecule	0
Ni	impropers	# of impropers Ni in molecule	0
Nf	fragments	# of fragments Nf in molecule	0
Ninteger Ndouble	body	# of integer and floating-point values in body particle	0
Mtotal	mass	total mass of molecule	computed
Xc Yc Zc	com	coordinates of center-of-mass of molecule	computed
Ixx Iyy Izz Ixy Ixz Iyz	inertia	6 components of inertia tensor of molecule	computed

For *mass*, *com*, and *inertia*, the default is for LAMMPS to calculate this quantity itself if needed, assuming the molecules consist of a set of point particles or finite-size particles (with a non-zero diameter) that do **not** overlap. If finite-size particles in the molecule **do** overlap, LAMMPS will not account for the overlap effects when calculating any of these 3 quantities, so you should pre-compute them yourself and list the values in the file.

The mass and center-of-mass coordinates (Xc,Yc,Zc) are self-explanatory. 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.

These are the allowed section keywords for the body of the file.

- *Coords, Types, Molecules, Fragments, Charges, Diameters, Dipoles, Masses* = atom-property sections
- *Bonds, Angles, Dihedrals, Impropers* = molecular topology sections
- *Special Bond Counts, Special Bonds* = special neighbor info
- *Shake Flags, Shake Atoms, Shake Bond Types* = SHAKE info
- *Body Integers, Body Doubles* = body-property sections

For the Types, Bonds, Angles, Dihedrals, and Impropers sections, each atom/bond/angle/etc type can be specified either as a number (numeric type) or as an alphanumeric type label. The latter is only allowed if type labels have been defined, either by the *labelmap* command or in data files read by the *read_data* command which have sections for Atom Type Labels, Bond Type Labels, Angle Type Labels, etc. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used. When using type labels, any values specified as *offset* are ignored.

If a Bonds section is specified then the Special Bond Counts and Special Bonds sections can also be used, if desired, to explicitly list the 1-2, 1-3, 1-4 neighbors within the molecule topology (see details below). This is optional since if these sections are not included, LAMMPS will auto-generate this information. Note that LAMMPS uses this info to properly exclude or weight bonded pairwise interactions between bonded atoms. See the *special_bonds* command for more details. One reason to list the special bond info explicitly is for the *thermalized Drude oscillator model* which treats the bonds between nuclear cores and Drude electrons in a different manner.

Note

Whether a section is required depends on how the molecule template is used by other LAMMPS commands. For example, to add a molecule via the *fix deposit* command, the Coords and Types sections are required. To add a rigid body via the *fix pour* command, the Bonds (Angles, etc) sections are not required, since the molecule will be treated as a rigid body. Some sections are optional. For example, the *fix pour* command can be used to add "molecules" which are clusters of finite-size granular particles. If the Diameters section is not specified, each

particle in the molecule will have a default diameter of 1.0. See the doc pages for LAMMPS commands that use molecule templates for more details.

Each section is listed below in alphabetic order. The format of each section is described including the number of lines it must contain and rules (if any) for whether it can appear in the data file. For per- atom sections, entries should be numbered from 1 to Natoms (where Natoms is the number of atoms in the template), indicating which atom (or bond, etc) the entry applies to. Per-atom sections need to include a setting for every atom, but the atoms can be listed in any order.

Coords section:

- one line per atom
- line syntax: ID x y z
- x,y,z = coordinate of atom

Types section:

- one line per atom
- line syntax: ID type
- type = atom type of atom (1-Natomtype, or type label)

Molecules section:

- one line per atom
- line syntax: ID molecule-ID
- molecule-ID = molecule ID of atom

Fragments section:

- one line per fragment
- line syntax: ID a b c d ...
- a,b,c,d,... = IDs of atoms in fragment

The ID of a fragment can only contain alphanumeric characters and underscores. The atom IDs should be values from 1 to Natoms, where Natoms = # of atoms in the molecule.

Charges section:

- one line per atom
- line syntax: ID q
- q = charge on atom

This section is only allowed for *atom styles* that support charge. If this section is not included, the default charge on each atom in the molecule is 0.0.

Diameters section:

- one line per atom
- line syntax: ID diam
- diam = diameter of atom

This section is only allowed for *atom styles* that support finite-size spherical particles, e.g. atom_style sphere. If not listed, the default diameter of each atom in the molecule is 1.0.

Added in version 7Feb2024.

Dipoles section:

- one line per atom
- line syntax: ID mux muy muz
- mux,muy,muz = x-, y-, and z-component of point dipole vector of atom

This section is only allowed for *atom styles* that support particles with point dipoles, e.g. atom_style dipole. If not listed, the default dipole component of each atom in the molecule is set to 0.0.

Masses section:

- one line per atom
- line syntax: ID mass
- mass = mass of atom

This section is only allowed for *atom styles* that support per-atom mass, as opposed to per-type mass. See the *mass* command for details. If this section is not included, the default mass for each atom is derived from its volume (see Diameters section) and a default density of 1.0, in *units* of mass/volume.

Bonds section:

- one line per bond
- line syntax: ID type atom1 atom2
- type = bond type (1-Nbondtype, or type label)
- atom1,atom2 = IDs of atoms in bond

The IDs for the two atoms in each bond should be values from 1 to Natoms, where Natoms = # of atoms in the molecule.

Angles section:

- one line per angle
- line syntax: ID type atom1 atom2 atom3
- type = angle type (1-Nangletype, or type label)
- atom1,atom2,atom3 = IDs of atoms in angle

The IDs for the three atoms in each angle should be values from 1 to Natoms, where Natoms = # of atoms in the molecule. The three atoms are ordered linearly within the angle. Thus the central atom (around which the angle is computed) is the atom2 in the list.

Dihedrals section:

- one line per dihedral
- line syntax: ID type atom1 atom2 atom3 atom4
- type = dihedral type (1-Ndihedraltype, or type label)
- atom1,atom2,atom3,atom4 = IDs of atoms in dihedral

The IDs for the four atoms in each dihedral should be values from 1 to Natoms, where Natoms = # of atoms in the molecule. The 4 atoms are ordered linearly within the dihedral.

Impropers section:

- one line per improper
- line syntax: ID type atom1 atom2 atom3 atom4
- type = improper type (1-Nimpropertype, or type label)
- atom1,atom2,atom3,atom4 = IDs of atoms in improper

The IDs for the four atoms in each improper should be values from 1 to Natoms, where Natoms = # of atoms in the molecule. The ordering of the 4 atoms determines the definition of the improper angle used in the formula for the defined *improper style*. See the doc pages for individual styles for details.

Special Bond Counts section:

- one line per atom
- line syntax: ID N1 N2 N3
- N1 = # of 1-2 bonds
- N2 = # of 1-3 bonds
- N3 = # of 1-4 bonds

N1, N2, N3 are the number of 1-2, 1-3, 1-4 neighbors respectively of this atom within the topology of the molecule. See the *special_bonds* page for more discussion of 1-2, 1-3, 1-4 neighbors. If this section appears, the Special Bonds section must also appear.

As explained above, LAMMPS will auto-generate this information if this section is not specified. If specified, this section will override what would be auto-generated.

Special Bonds section:

- one line per atom
- line syntax: ID a b c d ...
- a,b,c,d,... = IDs of atoms in N1+N2+N3 special bonds

A, b, c, d, etc are the IDs of the n1+n2+n3 atoms that are 1-2, 1-3, 1-4 neighbors of this atom. The IDs should be values from 1 to Natoms, where Natoms = # of atoms in the molecule. The first N1 values should be the 1-2 neighbors, the next N2 should be the 1-3 neighbors, the last N3 should be the 1-4 neighbors. No atom ID should appear more than once. See the *special_bonds* doc page for more discussion of 1-2, 1-3, 1-4 neighbors. If this section appears, the Special Bond Counts section must also appear.

As explained above, LAMMPS will auto-generate this information if this section is not specified. If specified, this section will override what would be auto-generated.

Shake Flags section:

- one line per atom
- line syntax: ID flag
- flag = 0,1,2,3,4

This section is only needed when molecules created using the template will be constrained by SHAKE via the "fix shake" command. The other two Shake sections must also appear in the file, following this one.

The meaning of the flag for each atom is as follows. See the *fix shake* page for a further description of SHAKE clusters.

- 0 = not part of a SHAKE cluster
- 1 = part of a SHAKE angle cluster (two bonds and the angle they form)
- 2 = part of a 2-atom SHAKE cluster with a single bond
- 3 = part of a 3-atom SHAKE cluster with two bonds
- 4 = part of a 4-atom SHAKE cluster with three bonds

Shake Atoms section:

- one line per atom
- line syntax: ID a b c d
- a,b,c,d = IDs of atoms in cluster

This section is only needed when molecules created using the template will be constrained by SHAKE via the "fix shake" command. The other two Shake sections must also appear in the file.

The a,b,c,d values are atom IDs (from 1 to Natoms) for all the atoms in the SHAKE cluster that this atom belongs to. The number of values that must appear is determined by the shake flag for the atom (see the Shake Flags section above). All atoms in a particular cluster should list their a,b,c,d values identically.

If flag = 0, no a,b,c,d values are listed on the line, just the (ignored) ID.

If flag = 1, a,b,c are listed, where a = ID of central atom in the angle, and b,c the other two atoms in the angle.

If flag = 2, a,b are listed, where a = ID of atom in bond with the lowest ID, and b = ID of atom in bond with the highest ID.

If flag = 3, a,b,c are listed, where a = ID of central atom, and b,c = IDs of other two atoms bonded to the central atom.

If flag = 4, a,b,c,d are listed, where a = ID of central atom, and b,c,d = IDs of other three atoms bonded to the central atom.

See the *fix shake* page for a further description of SHAKE clusters.

Shake Bond Types section:

- one line per atom
- line syntax: ID a b c
- a,b,c = bond types (or angle type) of bonds (or angle) in cluster

This section is only needed when molecules created using the template will be constrained by SHAKE via the "fix shake" command. The other two Shake sections must also appear in the file.

The a,b,c values are bond types for all bonds in the SHAKE cluster that this atom belongs to. Bond types may be either numbers (from 1 to Nbondtypes) or bond type labels as defined by the *labelmap* command or a "Bond Type Labels" section of a data file.

The number of values that must appear is determined by the shake flag for the atom (see the Shake Flags section above). All atoms in a particular cluster should list their a,b,c values identically.

If flag = 0, no a,b,c values are listed on the line, just the (ignored) ID.

If flag = 1, a,b,c are listed, where a = bondtype of the bond between the central atom and the first non-central atom (value b in the Shake Atoms section), b = bondtype of the bond between the central atom and the second non-central atom (value c in the Shake Atoms section), and c = the angle type (1 to Nangletypes, or angle type label) of the angle between the three atoms.

If flag = 2, only a is listed, where a = bondtype of the bond between the two atoms in the cluster.

If flag = 3, a,b are listed, where a = bondtype of the bond between the central atom and the first non-central atom (value b in the Shake Atoms section), and b = bondtype of the bond between the central atom and the second non-central atom (value c in the Shake Atoms section).

If flag = 4, a,b,c are listed, where a = bondtype of the bond between the central atom and the first non-central atom (value b in the Shake Atoms section), b = bondtype of the bond between the central atom and the second non-central atom (value c in the Shake Atoms section), and c = bondtype of the bond between the central atom and the third non-central atom (value d in the Shake Atoms section).

See the *fix shake* page for a further description of SHAKE clusters.

Body Integers section:

- one line
- line syntax: N E F
- N = number of sub-particles or number or vertices
- E,F = number of edges and faces

This section is only needed when the molecule is a body particle. the other Body section must also appear in the file.

The total number of values that must appear is determined by the body style, and must be equal to the Ninteger value given in the *body* header.

For *nparticle* and *rounded/polygon*, only the number of sub-particles or vertices N is required, and Ninteger should have a value of 1.

For *rounded/polyhedron*, the number of edges E and faces F is required, and Ninteger should have a value of 3.

See the *Howto body* page for a further description of the file format.

Body Doubles section:

- first line
- line syntax: Ixx Iyy Izz Ixy Ixz Iyz
- Ixx Iyy Izz Ixy Ixz Iyz = 6 components of inertia tensor of body particle
- one line per sub-particle or vertex
- line syntax: x y z
- x, y, z = coordinates of sub-particle or vertex

- one line per edge
- line syntax: N1 N2
- N1, N2 = vertex indices
- one line per face
- line syntax: N1 N2 N3 N4
- N1, N2, N3, N4 = vertex indices
- last line
- line syntax: diam
- diam = rounded diameter that surrounds each vertex

This section is only needed when the molecule is a body particle. the other Body section must also appear in the file.

The total number of values that must appear is determined by the body style, and must be equal to the Ndouble value given in the *body* header. The 6 moments of inertia and the 3N coordinates of the sub-particles or vertices are required for all body styles.

For *rounded/polygon*, the E = 6 + 3*N + 1 edges are automatically determined from the vertices.

For *rounded/polyhedron*, the 2E vertex indices for the end points of the edges and 4F vertex indices defining the faces are required.

See the *Howto body* page for a further description of the file format.

1.66.5 Restrictions

None

1.66.6 Related commands

fix deposit, *fix pour*, *fix gcmc*

1.66.7 Default

The default keywords values are offset 0 0 0 0 0 0 and scale = 1.0.

1.67 neb command

1.67.1 Syntax

```
neb etol ftol N1 N2 Nevery file-style arg keyword values
```

- etol = stopping tolerance for energy (dimensionless)
- ftol = stopping tolerance for force (force units)
- N1 = max # of iterations (timesteps) to run initial NEB

- N2 = max # of iterations (timesteps) to run barrier-climbing NEB
- Nevery = print replica energies and reaction coordinates every this many timesteps
- file-style = *final* or *each* or *none*

  final arg = filename
    filename = file with initial coords for final replica
      coords for intermediate replicas are linearly interpolated
      between first and last replica
  each arg = filename
    filename = unique filename for each replica (except first)
      with its initial coords
  none arg = no argument all replicas assumed to already have
      their initial coords

- zero or more keyword/value pairs may be appended
- keyword = *verbosity*

  verbosity value = verbose or default or terse
    verbose = very detailed per-replica output
    default = some per-replica output
    terse = only global state output

1.67.2 Examples

```
neb 0.1 0.0 1000 500 50 final coords.final
neb 0.0 0.001 1000 500 50 each coords.initial.$i
neb 0.0 0.001 1000 500 50 none verbose
```

1.67.3 Description

Perform a nudged elastic band (NEB) calculation using multiple replicas of a system. Two or more replicas must be used; the first and last are the end points of the transition path.

NEB is a method for finding both the atomic configurations and height of the energy barrier associated with a transition state, e.g. for an atom to perform a diffusive hop from one energy basin to another in a coordinated fashion with its neighbors. The implementation in LAMMPS follows the discussion in these 4 papers: *(HenkelmanA)*, *(HenkelmanB)*, *(Nakano)* and *(Maras)*.

Each replica runs on a partition of one or more processors. Processor partitions are defined at run-time using the *-partition command-line switch*. Note that if you have MPI installed, you can run a multi-replica simulation with more replicas (partitions) than you have physical processors, e.g you can run a 10-replica simulation on just one or two processors. You will simply not get the performance speed-up you would see with one or more physical processors per replica. See the *Howto replica* doc page for further discussion.

Note

As explained below, a NEB calculation performs a damped dynamics minimization across all the replicas. The minimizer uses whatever timestep you have defined in your input script, via the *timestep* command. Often NEB will converge more quickly if you use a timestep about 10x larger than you would normally use for dynamics simulations.

When a NEB calculation is performed, it is assumed that each replica is running the same system, though LAMMPS does not check for this. I.e. the simulation domain, the number of atoms, the interaction potentials, and the starting configuration when the neb command is issued should be the same for every replica.

In a NEB calculation each replica is connected to other replicas by inter-replica nudging forces. These forces are imposed by the *fix neb* command, which must be used in conjunction with the neb command. The group used to define the fix neb command defines the NEB atoms which are the only ones that inter-replica springs are applied to. If the group does not include all atoms, then non-NEB atoms have no inter-replica springs and the forces they feel and their motion is computed in the usual way due only to other atoms within their replica. Conceptually, the non-NEB atoms provide a background force field for the NEB atoms. They can be allowed to move during the NEB minimization procedure (which will typically induce different coordinates for non-NEB atoms in different replicas), or held fixed using other LAMMPS commands such as *fix setforce*. Note that the *partition* command can be used to invoke a command on a subset of the replicas, e.g. if you wish to hold NEB or non-NEB atoms fixed in only the end-point replicas.

The initial atomic configuration for each of the replicas can be specified in different manners via the *file-style* setting, as discussed below. Only atoms whose initial coordinates should differ from the current configuration need be specified.

Conceptually, the initial and final configurations for the first replica should be states on either side of an energy barrier.

As explained below, the initial configurations of intermediate replicas can be atomic coordinates interpolated in a linear fashion between the first and last replicas. This is often adequate for simple transitions. For more complex transitions, it may lead to slow convergence or even bad results if the minimum energy path (MEP, see below) of states over the barrier cannot be correctly converged to from such an initial path. In this case, you will want to generate initial states for the intermediate replicas that are geometrically closer to the MEP and read them in.

For a *file-style* setting of *final*, a filename is specified which contains atomic coordinates for zero or more atoms, in the format described below. For each atom that appears in the file, the new coordinates are assigned to that atom in the final replica. Each intermediate replica also assigns a new position to that atom in an interpolated manner. This is done by using the current position of the atom as the starting point and the read-in position as the final point. The distance between them is calculated, and the new position is assigned to be a fraction of the distance. E.g. if there are 10 replicas, the second replica will assign a position that is 10% of the distance along a line between the starting and final point, and the 9th replica will assign a position that is 90% of the distance along the line. Note that for this procedure to produce consistent coordinates across all the replicas, the current coordinates need to be the same in all replicas. LAMMPS does not check for this, but invalid initial configurations will likely result if it is not the case.

Note

The "distance" between the starting and final point is calculated in a minimum-image sense for a periodic simulation box. This means that if the two positions are on opposite sides of a box (periodic in that dimension), the distance between them will be small, because the periodic image of one of the atoms is close to the other. Similarly, even if the assigned position resulting from the interpolation is outside the periodic box, the atom will be wrapped back into the box when the NEB calculation begins.

For a *file-style* setting of *each*, a filename is specified which is assumed to be unique to each replica. This can be done by using a variable in the filename, e.g.

```
variable i equal part
neb 0.0 0.001 1000 500 50 each coords.initial.$i
```

which in this case will substitute the partition ID (0 to N-1) for the variable I, which is also effectively the replica ID. See the *variable* command for other options, such as using world-, universe-, or uloop-style variables.

Each replica (except the first replica) will read its file, formatted as described below, and for any atom that appears in the file, assign the specified coordinates to its atom. The various files do not need to contain the same set of atoms.

For a *file-style* setting of *none*, no filename is specified. Each replica is assumed to already be in its initial configuration at the time the neb command is issued. This allows each replica to define its own configuration by reading a replica-specific data or restart or dump file, via the *read_data*, *read_restart*, or *read_dump* commands. The replica-specific names of these files can be specified as in the discussion above for the *each* file-style. Also see the section below for how a NEB calculation can produce restart files, so that a long calculation can be restarted if needed.

Note

None of the *file-style* settings change the initial configuration of any atom in the first replica. The first replica must thus be in the correct initial configuration at the time the neb command is issued.

A NEB calculation proceeds in two stages, each of which is a minimization procedure, performed via damped dynamics. To enable this, you must first define a damped dynamics *min_style*, such as *quickmin* or *fire*. The *cg*, *sd*, and *hftn* styles cannot be used, since they perform iterative line searches in their inner loop, which cannot be easily synchronized across multiple replicas.

The minimizer tolerances for energy and force are set by *etol* and *ftol*, the same as for the *minimize* command.

A non-zero *etol* means that the NEB calculation will terminate if the energy criterion is met by every replica. The energies being compared to *etol* do not include any contribution from the inter-replica nudging forces, since these are non-conservative. A non-zero *ftol* means that the NEB calculation will terminate if the force criterion is met by every replica. The forces being compared to *ftol* include the inter-replica nudging forces.

The maximum number of iterations in each stage is set by *N1* and *N2*. These are effectively timestep counts since each iteration of damped dynamics is like a single timestep in a dynamics *run*. During both stages, the potential energy of each replica and its normalized distance along the reaction path (reaction coordinate RD) will be printed to the screen and log file every *Nevery* timesteps. The RD is 0 and 1 for the first and last replica. For intermediate replicas, it is the cumulative distance (normalized by the total cumulative distance) between adjacent replicas, where "distance" is defined as the length of the 3N-vector of differences in atomic coordinates, where N is the number of NEB atoms involved in the transition. These outputs allow you to monitor NEB's progress in finding a good energy barrier. *N1* and *N2* must both be multiples of *Nevery*.

In the first stage of NEB, the set of replicas should converge toward a minimum energy path (MEP) of conformational states that transition over a barrier. The MEP for a transition is defined as a sequence of 3N-dimensional states, each of which has a potential energy gradient parallel to the MEP itself. The configuration of highest energy along a MEP corresponds to a saddle point. The replica states will also be roughly equally spaced along the MEP due to the inter-replica nudging force added by the *fix neb* command.

In the second stage of NEB, the replica with the highest energy is selected and the inter-replica forces on it are converted to a force that drives its atom coordinates to the top or saddle point of the barrier, via the barrier-climbing calculation described in *(HenkelmanB)*. As before, the other replicas rearrange themselves along the MEP so as to be roughly equally spaced.

When both stages are complete, if the NEB calculation was successful, the configurations of the replicas should be along (close to) the MEP and the replica with the highest energy should be an atomic configuration at (close to) the saddle point of the transition. The potential energies for the set of replicas represents the energy profile of the transition along the MEP.

A few other settings in your input script are required or advised to perform a NEB calculation. See the NOTE about the choice of timestep at the beginning of this doc page.

An atom map must be defined which it is not by default for *atom_style atomic* problems. The *atom_modify map* command can be used to do this.

The minimizers in LAMMPS operate on all atoms in your system, even non-NEB atoms, as defined above. To prevent non-NEB atoms from moving during the minimization, you should use the *fix setforce* command to set the force on each of those atoms to 0.0. This is not required, and may not even be desired in some cases, but if those atoms move too far (e.g. because the initial state of your system was not well-minimized), it can cause problems for the NEB procedure.

The damped dynamics *minimizers*, such as *quickmin* and *fire*), adjust the position and velocity of the atoms via an Euler integration step. Thus you must define an appropriate *timestep* to use with NEB. As mentioned above, NEB will often converge more quickly if you use a timestep about 10x larger than you would normally use for dynamics simulations.

Each file read by the neb command containing atomic coordinates used to initialize one or more replicas must be formatted as follows.

The file can be ASCII text or a gzipped text file (detected by a .gz suffix). The file can contain initial blank lines or comment lines starting with "#" which are ignored. The first non-blank, non-comment line should list N = the number of lines to follow. The N successive lines contain the following information:

```
ID1 x1 y1 z1
ID2 x2 y2 z2
...
IDN xN yN zN
```

The fields are the atom ID, followed by the x,y,z coordinates. The lines can be listed in any order. Additional trailing information on the line is OK, such as a comment.

Note that for a typical NEB calculation you do not need to specify initial coordinates for very many atoms to produce differing starting and final replicas whose intermediate replicas will converge to the energy barrier. Typically only new coordinates for atoms geometrically near the barrier need be specified.

Also note there is no requirement that the atoms in the file correspond to the NEB atoms in the group defined by the *fix neb* command. Not every NEB atom need be in the file, and non-NEB atoms can be listed in the file.

Four kinds of output can be generated during a NEB calculation: energy barrier statistics, thermodynamic output by each replica, dump files, and restart files.

When running with multiple partitions (each of which is a replica in this case), the print-out to the screen and master log.lammps file contains a line of output, printed once every *Nevery* timesteps. The amount of information printed in this line can be selected with the *verbosity* keyword. Available options are *terse*, *default*, and *verbose*.

With the *terse* setting, it contains the timestep, the maximum force of a replica, the maximum force per atom (in any replica), potential gradients in the initial, final, and climbing replicas, the forward and backward energy barriers, the total reaction coordinate (RDT).

With the *default* setting, additionally the normalized reaction coordinate and potential energy of each replica are printed.

With the *verbose* setting, additional per-replica properties are printed: the "path angle" (pathangle), the angle between the 3N-length tangent vector and the 3N-length force vector at image *i* (angletangrad), the angle between the 3N-length energy gradient vector of replica *i* and that of replica *i*+1 (anglegrad), the norm of the energy gradient (gradV), the the two-norm of the 3N-length force vector (RepForce), and the maximum force component of any atom (MaxAtomForce).

The "maximum force per replica" is the two-norm of the 3N-length force vector for the atoms in each replica, maximized across replicas, which is what the *ftol* setting is checking against. In this case, N is all the atoms in each replica. The "maximum force per atom" is the maximum force component of any atom in any replica. The potential gradients are the two-norm of the 3N-length force vector solely due to the interaction potential i.e. without adding in inter-replica forces.

The "reaction coordinate" (RD) for each replica is the two-norm of the 3N-length vector of distances between its atoms and the preceding replica's atoms, added to the RD of the preceding replica. The RD of the first replica RD1 = 0.0;

the RD of the final replica RDN = RDT, the total reaction coordinate. The normalized RDs are divided by RDT, so that they form a monotonically increasing sequence from zero to one. When computing RD, N only includes the atoms being operated on by the fix neb command.

The forward (reverse) energy barrier is the potential energy of the highest replica minus the energy of the first (last) replica.

The "path angle" (pathangle) for the replica i which is the angle between the 3N-length vectors $(R_{i-1}-R_i)$ and $(R_{i+1}-R_i)$ (where R_i is the atomic coordinates of replica i). A "path angle" of 180 indicates that replicas i-1, i and i+1 are aligned. "angletangrad" is the angle between the 3N-length tangent vector and the 3N-length force vector at image i. The tangent vector is calculated as in *(HenkelmanA)* for all intermediate replicas and at R2 - R1 and RM - RM-1 for the first and last replica, respectively. "anglegrad" is the angle between the 3N-length energy gradient vector of replica i and that of replica i+1. It is not defined for the final replica and reads nan. gradV is the norm of the energy gradient of image i (∇V). ReplicaForce is the two-norm of the 3N-length force vector (including nudging forces) for replica i. MaxAtomForce is the maximum force component of any atom in replica i.

When a NEB calculation does not converge properly, the supplementary information can help understanding what is going wrong. For instance when the path angle becomes acute, the definition of tangent used in the NEB calculation is questionable and the NEB cannot may diverge *(Maras)*.

When running on multiple partitions, LAMMPS produces additional log files for each partition, e.g. log.lammps.0, log.lammps.1, etc. For a NEB calculation, these contain the thermodynamic output for each replica.

If *dump* commands in the input script define a filename that includes a *universe* or *uloop* style *variable*, then one dump file (per dump command) will be created for each replica. At the end of the NEB calculation, the final snapshot in each file will contain the sequence of snapshots that transition the system over the energy barrier. Earlier snapshots will show the convergence of the replicas to the MEP.

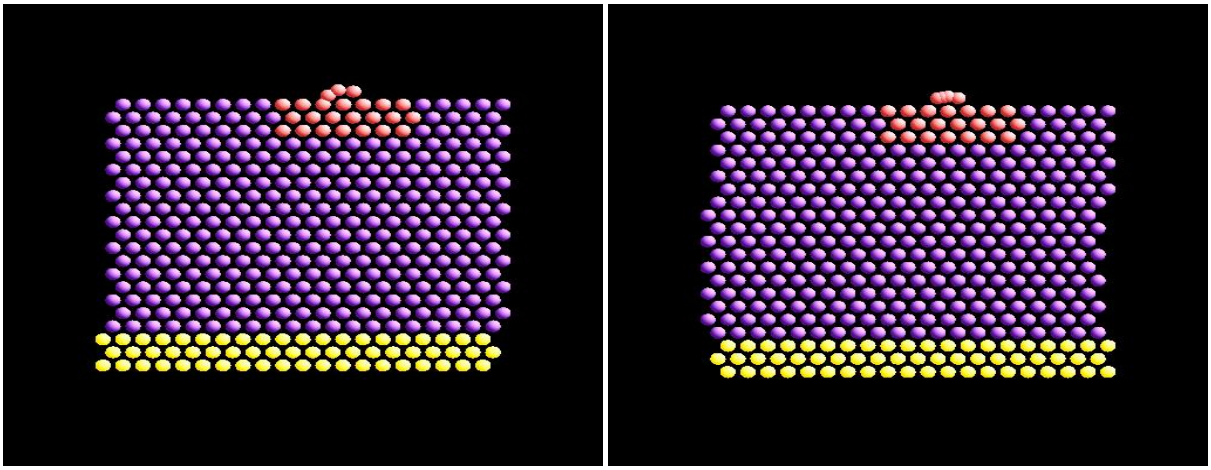
Likewise, *restart* filenames can be specified with a *universe* or *uloop* style *variable*, to generate restart files for each replica. These may be useful if the NEB calculation fails to converge properly to the MEP, and you wish to restart the calculation from an intermediate point with altered parameters.

There are 2 Python scripts provided in the tools/python directory, neb_combine.py and neb_final.py, which are useful in analyzing output from a NEB calculation. Assume a NEB simulation with M replicas, and the NEB atoms labeled with a specific atom type.

The neb_combine.py script extracts atom coords for the NEB atoms from all M dump files and creates a single dump file where each snapshot contains the NEB atoms from all the replicas and one copy of non-NEB atoms from the first replica (presumed to be identical in other replicas). This can be visualized/animated to see how the NEB atoms relax as the NEB calculation proceeds.

The neb_final.py script extracts the final snapshot from each of the M dump files to create a single dump file with M snapshots. This can be visualized to watch the system make its transition over the energy barrier.

To illustrate, here are images from the final snapshot produced by the neb_combine.py script run on the dump files produced by the two example input scripts in examples/neb.



1.67.4 Restrictions

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

1.67.5 Related commands

prd, *temper*, *fix langevin*, *fix viscous*, *fix neb*

1.67.6 Default

verbosity = default

(HenkelmanA) Henkelman and Jonsson, J Chem Phys, 113, 9978-9985 (2000).

(HenkelmanB) Henkelman, Uberuaga, Jonsson, J Chem Phys, 113, 9901-9904 (2000).

(Nakano) Nakano, Comp Phys Comm, 178, 280-289 (2008).

(Maras) Maras, Trushin, Stukowski, Ala-Nissila, Jonsson, Comp Phys Comm, 205, 13-21 (2016)

1.68 neb/spin command

1.68.1 Syntax

```
neb/spin etol ttol N1 N2 Nevery file-style arg keyword
```

- etol = stopping tolerance for energy (energy units)
- ttol = stopping tolerance for torque (units)
- N1 = max # of iterations (timesteps) to run initial NEB

- N2 = max # of iterations (timesteps) to run barrier-climbing NEB
- Nevery = print replica energies and reaction coordinates every this many timesteps
- file-style = *final* or *each* or *none*

  final arg = filename
    filename = file with initial coords for final replica
      coords for intermediate replicas are linearly interpolated
      between first and last replica
  each arg = filename
    filename = unique filename for each replica (except first)
      with its initial coords
  none arg = no argument all replicas assumed to already have
      their initial coords

- keyword = *verbose*

  verbose = print supplemental information

1.68.2 Examples

```
neb/spin 0.1 0.0 1000 500 50 final coords.final
neb/spin 0.0 0.001 1000 500 50 each coords.initial.$i
neb/spin 0.0 0.001 1000 500 50 none verbose
```

1.68.3 Description

Perform a geodesic nudged elastic band (GNEB) calculation using multiple replicas of a system. Two or more replicas must be used; the first and last are the end points of the transition path.

GNEB is a method for finding both the spin configurations and height of the energy barrier associated with a transition state, e.g. spins to perform a collective rotation from one energy basin to another. The implementation in LAMMPS follows the discussion in the following paper: *(Bessarab)*.

Each replica runs on a partition of one or more processors. Processor partitions are defined at run-time using the *-partition command-line switch*. Note that if you have MPI installed, you can run a multi-replica simulation with more replicas (partitions) than you have physical processors, e.g you can run a 10-replica simulation on just one or two processors. You will simply not get the performance speed-up you would see with one or more physical processors per replica. See the *Howto replica* doc page for further discussion.

Note

As explained below, a GNEB calculation performs a minimization across all the replicas. One of the *spin* style minimizers has to be defined in your input script.

When a GNEB calculation is performed, it is assumed that each replica is running the same system, though LAMMPS does not check for this. I.e. the simulation domain, the number of magnetic atoms, the interaction potentials, and the starting configuration when the neb command is issued should be the same for every replica.

In a GNEB calculation each replica is connected to other replicas by inter-replica nudging forces. These forces are imposed by the *fix neb/spin* command, which must be used in conjunction with the neb command. The group used to define the fix neb/spin command defines the GNEB magnetic atoms which are the only ones that inter-replica springs are applied to. If the group does not include all magnetic atoms, then non-GNEB magnetic atoms have no inter-replica

springs and the torques they feel and their precession motion is computed in the usual way due only to other magnetic atoms within their replica. Conceptually, the non-GNEB atoms provide a background force field for the GNEB atoms. Their magnetic spins can be allowed to evolve during the GNEB minimization procedure.

The initial spin configuration for each of the replicas can be specified in different manners via the *file-style* setting, as discussed below. Only atomic spins whose initial coordinates should differ from the current configuration need to be specified.

Conceptually, the initial and final configurations for the first replica should be states on either side of an energy barrier.

As explained below, the initial configurations of intermediate replicas can be spin coordinates interpolated in a linear fashion between the first and last replicas. This is often adequate for simple transitions. For more complex transitions, it may lead to slow convergence or even bad results if the minimum energy path (MEP, see below) of states over the barrier cannot be correctly converged to from such an initial path. In this case, you will want to generate initial states for the intermediate replicas that are geometrically closer to the MEP and read them in.

For a *file-style* setting of *final*, a filename is specified which contains atomic and spin coordinates for zero or more atoms, in the format described below. For each atom that appears in the file, the new coordinates are assigned to that atom in the final replica. Each intermediate replica also assigns a new spin to that atom in an interpolated manner. This is done by using the current direction of the spin at the starting point and the read-in direction as the final point. The "angular distance" between them is calculated, and the new direction is assigned to be a fraction of the angular distance.

Note

The "angular distance" between the starting and final point is evaluated in the geodesic sense, as described in *(Bessarab)*.

Note

The angular interpolation between the starting and final point is achieved using Rodrigues formula:

$$\vec{m}_i^{\nu} = \vec{m}_i^I \cos(\omega_i^{\nu}) + (\vec{k}_i \times \vec{m}_i^I) \sin(\omega_i^{\nu}) + (1.0 - \cos(\omega_i^{\nu}))\vec{k}_i(\vec{k}_i \cdot \vec{m}_i^I)$$

where $\vec{m}_i^I$ is the initial spin configuration for spin i, ω_i^{ν} is a rotation angle defined as:

$$\omega_i^{\nu} = (\nu - 1)\Delta\omega_i \text{ and } \Delta\omega_i = \frac{\omega_i}{Q - 1}$$

with ν the image number, Q the total number of images, and ω_i the total rotation between the initial and final spins. $\vec{k}_i$ defines a rotation axis such as:

$$\vec{k}_i = \frac{\vec{m}_i^I \times \vec{m}_i^F}{|\vec{m}_i^I \times \vec{m}_i^F|}$$

if the initial and final spins are not aligned. If the initial and final spins are aligned, then their cross product is null, and the expression above does not apply. If they point toward the same direction, the intermediate images conserve the same orientation. If the initial and final spins are aligned, but point toward opposite directions, an arbitrary rotation vector belonging to the plane perpendicular to initial and final spins is chosen. In this case, a warning message is displayed.

For a *file-style* setting of *each*, a filename is specified which is assumed to be unique to each replica. See the *neb* documentation page for more information about this option.

For a *file-style* setting of *none*, no filename is specified. Each replica is assumed to already be in its initial configuration at the time the neb command is issued. This allows each replica to define its own configuration by reading a replica-specific data or restart or dump file, via the *read_data*, *read_restart*, or *read_dump* commands. The replica-specific names of these files can be specified as in the discussion above for the *each* file-style. Also see the section below for how a NEB calculation can produce restart files, so that a long calculation can be restarted if needed.

Note

None of the *file-style* settings change the initial configuration of any atom in the first replica. The first replica must thus be in the correct initial configuration at the time the neb command is issued.

A NEB calculation proceeds in two stages, each of which is a minimization procedure. To enable this, you must first define a *min_style*, using either the *spin*, *spin/cg*, or *spin/lbfgs* style (see *min_spin* for more information). The other styles cannot be used, since they relax the lattice degrees of freedom instead of the spins.

The minimizer tolerances for energy and force are set by *etol* and *ttol*, the same as for the *minimize* command.

A non-zero *etol* means that the GNEB calculation will terminate if the energy criterion is met by every replica. The energies being compared to *etol* do not include any contribution from the inter-replica nudging forces, since these are non-conservative. A non-zero *ttol* means that the GNEB calculation will terminate if the torque criterion is met by every replica. The torques being compared to *ttol* include the inter-replica nudging forces.

The maximum number of iterations in each stage is set by *N1* and *N2*. These are effectively timestep counts since each iteration of damped dynamics is like a single timestep in a dynamics *run*. During both stages, the potential energy of each replica and its normalized distance along the reaction path (reaction coordinate RD) will be printed to the screen and log file every *Nevery* timesteps. The RD is 0 and 1 for the first and last replica. For intermediate replicas, it is the cumulative angular distance (normalized by the total cumulative angular distance) between adjacent replicas, where "distance" is defined as the length of the 3N-vector of the geodesic distances in spin coordinates, with N the number of GNEB spins involved (see equation (13) in *(Bessarab)*). These outputs allow you to monitor NEB's progress in finding a good energy barrier. *N1* and *N2* must both be multiples of *Nevery*.

In the first stage of GNEB, the set of replicas should converge toward a minimum energy path (MEP) of conformational states that transition over a barrier. The MEP for a transition is defined as a sequence of 3N-dimensional spin states, each of which has a potential energy gradient parallel to the MEP itself. The configuration of highest energy along a MEP corresponds to a saddle point. The replica states will also be roughly equally spaced along the MEP due to the inter-replica nudging force added by the *fix neb* command.

In the second stage of GNEB, the replica with the highest energy is selected and the inter-replica forces on it are converted to a force that drives its spin coordinates to the top or saddle point of the barrier, via the barrier-climbing calculation described in *(Bessarab)*. As before, the other replicas rearrange themselves along the MEP so as to be roughly equally spaced.

When both stages are complete, if the GNEB calculation was successful, the configurations of the replicas should be along (close to) the MEP and the replica with the highest energy should be a spin configuration at (close to) the saddle point of the transition. The potential energies for the set of replicas represents the energy profile of the transition along the MEP.

An atom map must be defined which it is not by default for *atom_style atomic* problems. The *atom_modify map* command can be used to do this.

An initial value can be defined for the timestep. Although, the *spin* minimization algorithm is an adaptive timestep methodology, so that this timestep is likely to evolve during the calculation.

The minimizers in LAMMPS operate on all spins in your system, even non-GNEB atoms, as defined above.

Each file read by the neb/spin command containing spin coordinates used to initialize one or more replicas must be formatted as follows.

The file can be ASCII text or a gzipped text file (detected by a .gz suffix). The file can contain initial blank lines or comment lines starting with "#" which are ignored. The first non-blank, non-comment line should list N = the number of lines to follow. The N successive lines contain the following information:

```
ID1 g1 x1 y1 z1 sx1 sy1 sz1
ID2 g2 x2 y2 z2 sx2 sy2 sz2
...
IDN gN yN zN sxN syN szN
```

The fields are the atom ID, the norm of the associated magnetic spin, followed by the *x,y,z* coordinates and the *sx,sy,sz* spin coordinates. The lines can be listed in any order. Additional trailing information on the line is OK, such as a comment.

Note that for a typical GNEB calculation you do not need to specify initial spin coordinates for very many atoms to produce differing starting and final replicas whose intermediate replicas will converge to the energy barrier. Typically only new spin coordinates for atoms geometrically near the barrier need be specified.

Also note there is no requirement that the atoms in the file correspond to the GNEB atoms in the group defined by the *fix neb* command. Not every GNEB atom need be in the file, and non-GNEB atoms can be listed in the file.

Four kinds of output can be generated during a GNEB calculation: energy barrier statistics, thermodynamic output by each replica, dump files, and restart files.

When running with multiple partitions (each of which is a replica in this case), the print-out to the screen and master log.lammps file contains a line of output, printed once every *Nevery* timesteps. It contains the timestep, the maximum torque per replica, the maximum torque per atom (in any replica), potential gradients in the initial, final, and climbing replicas, the forward and backward energy barriers, the total reaction coordinate (RDT), and the normalized reaction coordinate and potential energy of each replica.

The "maximum torque per replica" is the two-norm of the 3N-length vector given by the cross product of a spin by its precession vector omega, in each replica, maximized across replicas, which is what the *ttol* setting is checking against. In this case, N is all the atoms in each replica. The "maximum torque per atom" is the maximum torque component of any atom in any replica. The potential gradients are the two-norm of the 3N-length magnetic precession vector solely due to the interaction potential i.e. without adding in inter-replica forces, and projected along the path tangent (as detailed in Appendix D of *(Bessarab)*).

The "reaction coordinate" (RD) for each replica is the two-norm of the 3N-length vector of geodesic distances between its spins and the preceding replica's spins (see equation (13) of *(Bessarab)*), added to the RD of the preceding replica. The RD of the first replica RD1 = 0.0; the RD of the final replica RDN = RDT, the total reaction coordinate. The normalized RDs are divided by RDT, so that they form a monotonically increasing sequence from zero to one. When computing RD, N only includes the spins being operated on by the fix neb/spin command.

The forward (reverse) energy barrier is the potential energy of the highest replica minus the energy of the first (last) replica.

Supplementary information for all replicas can be printed out to the screen and master log.lammps file by adding the *verbose* keyword. This information include the following. The "GradVidottan" are the projections of the potential gradient for the replica i on its tangent vector (as detailed in Appendix D of *(Bessarab)*). The "DNi" are the non normalized geodesic distances (see equation (13) of *(Bessarab)*), between a replica i and the next replica i+1. For the last replica, this distance is not defined and a "NAN" value is the corresponding output.

When a NEB calculation does not converge properly, the supplementary information can help understanding what is going wrong.

When running on multiple partitions, LAMMPS produces additional log files for each partition, e.g. log.lammps.0, log.lammps.1, etc. For a GNEB calculation, these contain the thermodynamic output for each replica.

If *dump* commands in the input script define a filename that includes a *universe* or *uloop* style *variable*, then one dump file (per dump command) will be created for each replica. At the end of the GNEB calculation, the final snapshot in each file will contain the sequence of snapshots that transition the system over the energy barrier. Earlier snapshots will show the convergence of the replicas to the MEP.

Likewise, *restart* filenames can be specified with a *universe* or *uloop* style *variable*, to generate restart files for each replica. These may be useful if the GNEB calculation fails to converge properly to the MEP, and you wish to restart the calculation from an intermediate point with altered parameters.

A c file script in provided in the tool/spin/interpolate_gneb directory, that interpolates the MEP given the information provided by the *verbose* output option (as detailed in Appendix D of *(Bessarab)*).

1.68.4 Restrictions

This command can only be used if LAMMPS was built with the SPIN package. See the *Build package* doc page for more info.

For magnetic GNEB calculations, only the *spin_none* value for the *line* keyword can be used when minimization styles *spin/cg* and *spin/lbfgs* are employed.

1.68.5 Related commands

min/spin, *fix neb/spin*

1.68.6 Default

none

(Bessarab) Bessarab, Uzdin, Jonsson, Comp Phys Comm, 196, 335-347 (2015).

1.69 neigh_modify command

1.69.1 Syntax

```
neigh_modify keyword values ...
```

- one or more keyword/value pairs may be listed

```
keyword = delay or every or check or once or cluster or include or exclude or page or one or binsize
  or collection/type or collection/interval
  delay value = N
    N = delay building neighbor lists until this many steps since last build
  every value = M
    M = consider building neighbor lists every this many steps
```

check value = yes or no
  yes = only build if at least one atom has moved half the skin distance or more
  no = always build on 1st step where every and delay are conditions are satisfied
once value = yes or no
  yes = only build neighbor list once at start of run and never rebuild
  no = rebuild neighbor list according to other settings
cluster value = yes or no
  yes = check bond,angle,etc neighbor list for nearby clusters
  no = do not check bond,angle,etc neighbor list for nearby clusters
include value = group-ID
  group-ID = only build pair neighbor lists for atoms in this group
exclude values:
  type M N
    M,N = exclude if one atom in pair is type M, other is type N (M and N may be type labels)
  group group1-ID group2-ID
    group1-ID,group2-ID = exclude if one atom is in 1st group, other in 2nd
  molecule/intra group-ID
    group-ID = exclude if both atoms are in the same molecule and in group
  molecule/inter group-ID
    group-ID = exclude if both atoms are in different molecules and in group
  none
    delete all exclude settings
page value = N
  N = number of pairs stored in a single neighbor page
one value = N
  N = max number of neighbors of one atom
binsize value = size
  size = bin size for neighbor list construction (distance units)
collection/type values = N arg1 ... argN
  N = number of custom collections
  arg = N separate lists of types (see below)
collection/interval values = N arg1 ... argN
  N = number of custom collections
  arg = N separate cutoffs for intervals (see below)

1.69.2 Examples

```
neigh_modify every 2 delay 10 check yes page 100000
neigh_modify exclude type 2 3
neigh_modify exclude group frozen frozen check no
neigh_modify exclude group residue1 chain3
neigh_modify exclude molecule/intra rigid
neigh_modify collection/type 2 1*2,5 3*4
neigh_modify collection/interval 2 1.0 10.0
```

1.69.3 Description

This command sets parameters that affect the building and use of pairwise neighbor lists. Depending on what pair interactions and other commands are defined, a simulation may require one or more neighbor lists.

The *every*, *delay*, *check*, and *once* options affect how often lists are built as a simulation runs. The *delay* setting means never build new lists until at least N steps after the previous build. The *every* setting means attempt to build lists every M steps (after the delay has passed). If the *check* setting is *no*, the lists are built on the first step that satisfies the *delay* and *every* settings. If the *check* setting is *yes*, then the *every* and *delay* settings determine when a build may possibly be performed, but an actual build only occurs if at least one atom has moved more than half the neighbor skin distance (specified in the *neighbor* command) since the last neighbor list build.

Impact of neighbor list settings

The choice of neighbor list settings can have a significant impact on the (parallel) performance of LAMMPS and the correctness of the simulation results. Since building the neighbor lists is time consuming, doing it less frequently can speed up a calculation. If the lists are rebuilt too infrequently, however, interacting pairs may be missing and thus the resulting pairwise interactions incorrect. The optimal settings depend on many factors like the properties of the simulated system (density, geometry, topology, temperature, pressure), the force field parameters and settings, the size of the timestep, neighbor list skin distance and more. The default settings are chosen to be very conservative to guarantee correctness of the simulation. They depend on the *check* flag heuristics to reduce the number of neighbor list rebuilds at a minor expense for executing the check. Determining the correctness of a specific choice of neighbor list settings is complicated by the fact that a neighbor list rebuild changes the order in which pairwise interactions are computed and thus - due to the limitations of floating-point math - the trajectory.

If the *once* setting is yes, then the neighbor list is only built once at the beginning of each run, and never rebuilt, except on steps when a restart file is written, or steps when a fix forces a rebuild to occur (e.g. fixes that create or delete atoms, such as *fix deposit* or *fix evaporate*). This setting should only be made if you are certain atoms will not move far enough that the neighbor list should be rebuilt, e.g. running a simulation of a cold crystal. Note that it is not that expensive to check if neighbor lists should be rebuilt.

When the rRESPA integrator is used (see the *run_style* command), the *every* and *delay* parameters refer to the longest (outermost) timestep.

The *cluster* option does a sanity test every time neighbor lists are built for bond, angle, dihedral, and improper interactions, to check that each set of 2, 3, or 4 atoms is a cluster of nearby atoms. It does this by computing the distance between pairs of atoms in the interaction and ensuring they are not further apart than half the periodic box length. If they are, an error is generated, since the interaction would be computed between far-away atoms instead of their nearby periodic images. The only way this should happen is if the pairwise cutoff is so short that atoms that are part of the same interaction are not communicated as ghost atoms. This is an unusual model (e.g. no pair interactions at all) and the problem can be fixed by use of the *comm_modify cutoff* command. Note that to save time, the default *cluster* setting is *no*, so that this check is not performed.

The *include* option limits the building of pairwise neighbor lists to atoms in the specified group. This can be useful for models where a large portion of the simulation is particles that do not interact with other particles or with each other via pairwise interactions. The group specified with this option must also be specified via the *atom_modify first* command. Note that specifying "all" as the group-ID effectively turns off the *include* option.

The *exclude* option turns off pairwise interactions between certain pairs of atoms, by not including them in the neighbor list. These are sample scenarios where this is useful:

- In crack simulations, pairwise interactions can be shut off between 2 slabs of atoms to effectively create a crack.
- When a large collection of atoms is treated as frozen, interactions between those atoms can be turned off to save needless computation. E.g. Using the *fix setforce* command to freeze a wall or portion of a bio-molecule.

- When one or more rigid bodies are specified, interactions within each body can be turned off to save needless computation. See the *fix rigid* command for more details.

Changed in version 29Aug2024: Support for type labels was added.

The *exclude type* option turns off the pairwise interaction if one atom is of type M and the other of type N. M can equal N. The *exclude group* option turns off the interaction if one atom is in the first group and the other is the second. Group1-ID can equal group2-ID. The *exclude molecule/intra* option turns off the interaction if both atoms are in the specified group and in the same molecule, as determined by their molecule ID. The *exclude molecule/inter* turns off the interaction between pairs of atoms that have different molecule IDs and are both in the specified group.

Each of the exclude options can be specified multiple times. The *exclude type* option is the most efficient option to use; it requires only a single check, no matter how many times it has been specified. The other exclude options are more expensive if specified multiple times; they require one check for each time they have been specified.

Note that the exclude options only affect pairwise interactions; see the *delete_bonds* command for information on turning off bond interactions.

Note

Excluding pairwise interactions will not work correctly when also using a long-range solver via the *kspace_style* command. LAMMPS will give a warning to this effect. This is because the short-range pairwise interaction needs to subtract off a term from the total energy for pairs whose short-range interaction is excluded, to compensate for how the long-range solver treats the interaction. This is done correctly for pairwise interactions that are excluded (or weighted) via the *special_bonds* command. But it is not done for interactions that are excluded via these neigh_modify exclude options.

The *page* and *one* options affect how memory is allocated for the neighbor lists. For most simulations the default settings for these options are fine, but if a very large problem is being run or a very long cutoff is being used, these parameters can be tuned. The indices of neighboring atoms are stored in "pages", which are allocated one after another as they fill up. The size of each page is set by the *page* value. A new page is allocated when the next atom's neighbors could potentially overflow the list. This threshold is set by the *one* value which tells LAMMPS the maximum number of neighbor's one atom can have.

Note

LAMMPS can crash without an error message if the number of neighbors for a single particle is larger than the *page* setting, which means it is much, much larger than the *one* setting. This is because LAMMPS does not error check these limits for every pairwise interaction (too costly), but only after all the particle's neighbors have been found. This problem usually means something is very wrong with the way you have setup your problem (particle spacing, cutoff length, neighbor skin distance, etc). If you really expect that many neighbors per particle, then boost the *one* and *page* settings accordingly.

The *binsize* option allows you to specify what size of bins will be used in neighbor list construction to sort and find neighboring atoms. By default, for *neighbor style bin*, LAMMPS uses bins that are 1/2 the size of the maximum pair cutoff. For *neighbor style multi*, the bins are 1/2 the size of the collection interaction cutoff. Typically these are good values for minimizing the time for neighbor list construction. This setting overrides the default. If you make it too big, there is little overhead due to looping over bins, but more atoms are checked. If you make it too small, the optimal number of atoms is checked, but bin overhead goes up. If you set the binsize to 0.0, LAMMPS will use the default binsize of 1/2 the cutoff.

The *collection/type* option allows you to define collections of atom types, used by the *multi* neighbor mode. By grouping atom types with similar physical size or interaction cutoff lengths, one may be able to improve performance by reducing overhead. You must first specify the number of collections N to be defined followed by N lists of types. Each list consists

of a series of type ranges separated by commas. The range can be specified as a single numeric value, or a wildcard asterisk can be used to specify a range of values. This takes the form "*" or "*n" or "n*" or "m*n". For example, if M = the number of atom types, then an asterisk with no numeric values means all types from 1 to M. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to M (inclusive). A middle asterisk means all types from m to n (inclusive). Note that all atom types must be included in exactly one of the N collections.

The *collection/interval* option provides a similar capability. This command allows a user to define collections by specifying a series of cutoff intervals. LAMMPS will automatically sort atoms into these intervals based on their type-dependent cutoffs or their finite size. You must first specify the number of collections N to be defined followed by N values representing the upper cutoff of each interval. This command is particularly useful for granular pair styles where the interaction distance of particles depends on their radius and may not depend on their atom type.

1.69.4 Restrictions

If the *delay* setting is non-zero, then it must be a multiple of the *every* setting.

The *molecule/intra* and *molecule/inter* exclusion options can only be used with atom styles that define molecule IDs.

The value of the *page* setting must be at least 10x larger than the *one* setting. This ensures neighbor pages are not mostly empty space.

The *exclude group* setting is currently not compatible with dynamic groups.

1.69.5 Related commands

neighbor, *delete_bonds*

1.69.6 Default

The option defaults are delay = 0, every = 1, check = yes, once = no, cluster = no, include = all (same as no include option defined), exclude = none, page = 100000, one = 2000, and binsize = 0.0.

1.70 neighbor command

1.70.1 Syntax

```
neighbor skin style
```

- skin = extra distance beyond force cutoff (distance units)
- style = *bin* or *nsq* or *multi* or *multi/old*

1.70.2 Examples

```
neighbor 0.3 bin
neighbor 2.0 nsq
```

1.70.3 Description

This command sets parameters that affect the building of pairwise neighbor lists. All atom pairs within a neighbor cutoff distance equal to the their force cutoff plus the *skin* distance are stored in the list. Typically, the larger the skin distance, the less often neighbor lists need to be built, but more pairs must be checked for possible force interactions every timestep. The default value for *skin* depends on the choice of units for the simulation; see the default values below.

The *skin* distance is also used to determine how often atoms migrate to new processors if the *check* option of the *neigh_modify* command is set to *yes*. Atoms are migrated (communicated) to new processors on the same timestep that neighbor lists are re-built.

The *style* value selects what algorithm is used to build the list. The *bin* style creates the list by binning which is an operation that scales linearly with N/P, the number of atoms per processor where N = total number of atoms and P = number of processors. It is almost always faster than the *nsq* style which scales as (N/P)^2. For unsolvated small molecules in a non-periodic box, the *nsq* choice can sometimes be faster. Either style should give the same answers.

The *multi* style is a modified binning algorithm that is useful for systems with a wide range of cutoff distances, e.g. due to different size particles. For granular pair styles, cutoffs are set to the sum of the maximum atomic radii for each atom type. For the *bin* style, the bin size is set to 1/2 of the largest cutoff distance between any pair of atom types and a single set of bins is defined to search over for all atom types. This can be inefficient if one pair of types has a very long cutoff, but other type pairs have a much shorter cutoff. The *multi* style uses different sized bins for collections of different sized particles, where "size" may mean the physical size of the particle or its cutoff distance for interacting with other particles. Different sets of bins are then used to construct the neighbor lists as as further described by Shire, Hanley, and Stratford *(Shire)* and Monti et al. *(Monti)*. This imposes some extra setup overhead, but the searches themselves may be much faster.

For instance in a dense binary system in d-dimensions with a ratio of the size of the largest to smallest collection bin λ, the computational costs of building a default neighbor list grows as λ^{2d} while the costs for *multi* grows as λ^{d}, equivalent to the cost of force evaluations, as argued in Monti et al. *(Monti)*. In other words, the neighboring costs of *multi* are expected to scale the same as force calculations, such that its relative cost is independent of the particle size ratio. This is not the case for the default style which becomes substantially more expensive with increasing size ratios.

By default in *multi*, each atom type defines a separate collection of particles. For systems where two or more atom types have the same size (either physical size or cutoff distance), the definition of collections can be customized, which can result in less overhead and faster performance. See the *neigh_modify* command for how to define custom collections. Whether the collection definition is customized or not, also see the *comm_modify mode multi* command for communication options that further improve performance in a manner consistent with neighbor style multi.

An alternate style, *multi/old*, sets the bin size to 1/2 of the shortest cutoff distance and multiple sets of bins are defined to search over for different atom types. This algorithm used to be the default *multi* algorithm in LAMMPS but was found to be significantly slower than the new approach. For the dense binary system, computational costs still grew as λ^{2d} at large enough λ. This is equivalent to the default style, albeit with a smaller prefactor. For now we are keeping the old option in case there are use cases where multi/old outperforms the new multi style.

Note

If there are multiple sub-styles in a *hybrid/overlay pair style* that cover the same atom types, but have significantly different cutoffs, the *multi* style does not apply. Instead, the *pair_modify neigh/trim* setting applies (which is *yes* by

default). Please check the neighbor list summary printed at the beginning of a calculation to verify that the desired set of neighbor list builds is performed.

The *neigh_modify* command has additional options that control how often neighbor lists are built and which pairs are stored in the list.

When a run is finished, counts of the number of neighbors stored in the pairwise list and the number of times neighbor lists were built are printed to the screen and log file. See the *Run output* page for details.

1.70.4 Restrictions

none

1.70.5 Related commands

neigh_modify, *units*, *comm_modify*

1.70.6 Default

0.3 bin for units = lj, skin = 0.3 sigma
2.0 bin for units = real or metal, skin = 2.0 Angstroms
0.001 bin for units = si, skin = 0.001 meters = 1.0 mm
0.1 bin for units = cgs, skin = 0.1 cm = 1.0 mm

(Shire) Shire, Hanley and Stratford, Comp. Part. Mech., (2020).

(Monti) Monti, Clemmer, Srivastava, Silbert, Grest, and Lechman, Phys. Rev. E, (2022).

1.71 newton command

1.71.1 Syntax

```
newton flag
newton flag1 flag2
```

- flag = *on* or *off* for both pairwise and bonded interactions
- flag1 = *on* or *off* for pairwise interactions
- flag2 = *on* or *off* for bonded interactions

1.71.2 Examples

```
newton off
newton on off
```

1.71.3 Description

This command turns Newton's third law *on* or *off* for pairwise and bonded interactions. For most problems, setting Newton's third law to *on* means a modest savings in computation at the cost of two times more communication. Whether this is faster depends on problem size, force cutoff lengths, a machine's compute/communication ratio, and how many processors are being used.

Setting the pairwise newton flag to *off* means that if two interacting atoms are on different processors, both processors compute their interaction and the resulting force information is not communicated. Similarly, for bonded interactions, newton *off* means that if a bond, angle, dihedral, or improper interaction contains atoms on 2 or more processors, the interaction is computed by each processor.

LAMMPS should produce the same answers for any newton flag settings, except for round-off issues.

With *run_style respa* and only bonded interactions (bond, angle, etc) computed in the innermost timestep, it may be faster to turn newton *off* for bonded interactions, to avoid extra communication in the innermost loop.

1.71.4 Restrictions

The newton bond setting cannot be changed after the simulation box is defined by a *read_data* or *create_box* command.

1.71.5 Related commands

run_style respa

1.71.6 Default

```
newton on
```

1.72 next command

1.72.1 Syntax

```
next variables
```

- variables = one or more variable names

1.72.2 Examples

```
next x
next a t x myTemp
```

1.72.3 Description

This command is used with variables defined by the *variable* command. It assigns the next value to the variable from the list of values defined for that variable by the *variable* command. Thus when that variable is subsequently substituted for in an input script command, the new value is used.

See the *variable* command for info on how to define and use different kinds of variables in LAMMPS input scripts. If a variable name is a single lower-case character from "a" to "z", it can be used in an input script command as $a or $z. If it is multiple letters, it can be used as ${myTemp}.

If multiple variables are used as arguments to the *next* command, then all must be of the same variable style: *index*, *loop*, *file*, *universe*, or *uloop*. An exception is that *universe*- and *uloop*-style variables can be mixed in the same *next* command.

All the variables specified with the next command are incremented by one value from their respective list of values. A *file*-style variable reads the next line from its associated file. An *atomfile*-style variable reads the next set of lines (one per atom) from its associated file. *String*- or *atom*- or *equal*- or *world*-style variables cannot be used with the next command, since they only store a single value.

When any of the variables in the next command has no more values, a flag is set that causes the input script to skip the next *jump* command encountered. This enables a loop containing a next command to exit. As explained in the *variable* command, the variable that has exhausted its values is also deleted. This allows it to be used and re-defined later in the input script. *File*-style and *atomfile*-style variables are exhausted when the end-of-file is reached.

When the next command is used with *index*- or *loop*-style variables, the next value is assigned to the variable for all processors. When the next command is used with *file*-style variables, the next line is read from its file and the string assigned to the variable. When the next command is used with *atomfile*-style variables, the next set of per-atom values is read from its file and assigned to the variable.

When the next command is used with *universe*- or *uloop*-style variables, all *universe*- or *uloop*-style variables must be listed in the next command. This is because of the manner in which the incrementing is done, using a single lock file for all variables. The next value (for each variable) is assigned to whichever processor partition executes the command first. All processors in the partition are assigned the same value(s). Running LAMMPS on multiple partitions of processors via the *-partition command-line switch*. *Universe*- and *uloop*-style variables are incremented using the files "tmp.lammps.variable" and "tmp.lammps.variable.lock" which you will see in your directory during and after such a LAMMPS run.

Here is an example of running a series of simulations using the next command with an *index*-style variable. If this input script is named in.polymer, 8 simulations would be run using data files from directories run1 through run8.

```
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
```

If the variable "d" were of style *universe*, and the same in.polymer input script were run on 3 partitions of processors, then the first 3 simulations would begin, one on each set of processors. Whichever partition finished first, it would

assign variable "d" the fourth value and run another simulation, and so forth until all 8 simulations were finished.

Jump and next commands can also be nested to enable multi-level loops. For example, this script will run 15 simulations in a double loop.

```
variable i loop 3
  variable j loop 5
  clear
  ...
  read_data data.polymer.$i$j
  print Running simulation $i.$j
  run 10000
  next j
  jump in.script
next i
jump in.script
```

Here is an example of a double loop which uses the *if* and *jump* commands to break out of the inner loop when a condition is met, then continues iterating through the outer loop.

```
label       loopa
variable    a loop 5
  label     loopb
  variable  b loop 5
  print     "A,B = $a,$b"
  run       10000
  if        $b > 2 then "jump in.script break"
  next      b
  jump      in.script loopb
label       break
variable    b delete

next        a
jump        in.script loopa
```

1.72.4 Restrictions

As described above.

1.72.5 Related commands

jump, *include*, *shell*, *variable*,

1.72.6 Default

none

1.73 package command

1.73.1 Syntax

```
package style args
```

- style = *gpu* or *intel* or *kokkos* or *omp*
- args = arguments specific to the style

  gpu args = Ngpu keyword value ...
    Ngpu = # of GPUs per node
    zero or more keyword/value pairs may be appended
    keywords = neigh or newton or pair/only or binsize or split or gpuID or tpa or blocksize or omp or platform or device_type or ocl_args
      neigh value = yes or no
        yes = neighbor list build on GPU (default)
        no = neighbor list build on CPU
      newton = off or on
        off = set Newton pairwise flag off (default and required)
        on = set Newton pairwise flag on (currently not allowed)
      pair/only = off or on
        off = apply "gpu" suffix to all available styles in the GPU package (default)
        on = apply "gpu" suffix only pair styles
      binsize value = size
        size = bin size for neighbor list construction (distance units)
      split = fraction
        fraction = fraction of atoms assigned to GPU (default = 1.0)
      tpa value = Nlanes
        Nlanes = # of GPU vector lanes (CUDA threads) used per atom
      blocksize value = size
        size = thread block size for pair force computation
      omp value = Nthreads
        Nthreads = number of OpenMP threads to use on CPU (default = 0)
      platform value = id
        id = For OpenCL, platform ID for the GPU or accelerator
      gpuID values = id
        id = ID of first GPU to be used on each node
      device_type value = intelgpu or nvidiagpu or amdgpu or applegpu or generic or custom,val1,val2,...
        val1,val2,... = custom OpenCL accelerator configuration parameters (see below for details)
      ocl_args value = args
        args = List of additional OpenCL compiler arguments delimited by colons
  intel args = NPhi keyword value ...

```
    Nphi = # of co-processors per node
    zero or more keyword/value pairs may be appended
    keywords = mode or omp or lrt or balance or ghost or tpc or tptask or pppm_table or no_affinity
      mode value = single or mixed or double
        single = perform force calculations in single precision
        mixed = perform force calculations in mixed precision
        double = perform force calculations in double precision
      omp value = Nthreads
        Nthreads = number of OpenMP threads to use on CPU (default = 0)
      lrt value = yes or no
        yes = use additional thread dedicated for some PPPM calculations
        no = do not dedicate an extra thread for some PPPM calculations
      balance value = split
        split = fraction of work to offload to co-processor, -1 for dynamic
      ghost value = yes or no
        yes = include ghost atoms for offload
        no = do not include ghost atoms for offload
      tpc value = Ntpc
        Ntpc = max number of co-processor threads per co-processor core (default = 4)
      tptask value = Ntptask
        Ntptask = max number of co-processor threads per MPI task (default = 240)
      pppm_table value = yes or no
        yes = Precompute pppm values in table (doesn't change accuracy)
        no = Compute pppm values on the fly
      no_affinity values = none
  kokkos args = keyword value ...
    zero or more keyword/value pairs may be appended
    keywords = neigh or neigh/qeq or neigh/thread or neigh/transpose or newton or binsize or comm␣
  ↪or comm/exchange or comm/forward or comm/pair/forward or comm/fix/forward or comm/
  ↪reverse or comm/pair/reverse or sort or atom/map or gpu/aware or pair/only
      neigh value = full or half
        full = full neighbor list
        half = half neighbor list built in thread-safe manner
      neigh/qeq value = full or half
        full = full neighbor list
        half = half neighbor list built in thread-safe manner
      neigh/thread value = off or on
        off = thread only over atoms
        on = thread over both atoms and neighbors
      neigh/transpose value = off or on
        off = use same memory layout for GPU neigh list build as pair style
        on = use transposed memory layout for GPU neigh list build
      newton = off or on
        off = set Newton pairwise and bonded flags off
        on = set Newton pairwise and bonded flags on
      binsize value = size
        size = bin size for neighbor list construction (distance units)
      comm value = no or host or device
        use value for comm/exchange and comm/forward and comm/pair/forward and comm/fix/
  ↪forward and comm/reverse
      comm/exchange value = no or host or device
      comm/forward value = no or host or device
      comm/pair/forward value = no or device
      comm/fix/forward value = no or device
```

```
    comm/reverse value = no or host or device
      no = perform communication pack/unpack in non-KOKKOS mode
      host = perform pack/unpack on host (e.g. with OpenMP threading)
      device = perform pack/unpack on device (e.g. on GPU)
    comm/pair/reverse value = no or device
      no = perform communication pack/unpack in non-KOKKOS mode
      device = perform pack/unpack on device (e.g. on GPU)
    sort value = no or device
      no = perform atom sorting in non-KOKKOS mode
      device = perform atom sorting on device (e.g. on GPU)
    atom/map value = no or device
      no = build atom map in non-KOKKOS mode
      device = build atom map on device (e.g. on GPU)
    gpu/aware = off or on
      off = do not use GPU-aware MPI
      on = use GPU-aware MPI (default)
    pair/only = off or on
      off = use device acceleration (e.g. GPU) for all available styles in the KOKKOS package␣
↪(default)
      on  = use device acceleration only for pair styles (and host acceleration for others)
  omp args = Nthreads keyword value ...
    Nthreads = # of OpenMP threads to associate with each MPI process
    zero or more keyword/value pairs may be appended
    keywords = neigh
      neigh value = yes or no
        yes = threaded neighbor list build (default)
        no = non-threaded neighbor list build
```

1.73.2 Examples

```
package gpu 0
package gpu 1 split 0.75
package gpu 2 split -1.0
package gpu 0 omp 2 device_type intelgpu
package kokkos neigh half comm device
package omp 0 neigh no
package omp 4
package intel 1
package intel 2 omp 4 mode mixed balance 0.5
```

1.73.3 Description

This command invokes package-specific settings for the various accelerator packages available in LAMMPS. Currently the following packages use settings from this command: GPU, INTEL, KOKKOS, and OPENMP.

If this command is specified in an input script, it must be near the top of the script, before the simulation box has been defined. This is because it specifies settings that the accelerator packages use in their initialization, before a simulation is defined.

This command can also be specified from the command-line when launching LAMMPS, using the "-pk" *command-line switch*. The syntax is exactly the same as when used in an input script.

Note that all of the accelerator packages require the package command to be specified (except the OPT package), if the package is to be used in a simulation (LAMMPS can be built with an accelerator package without using it in a particular simulation). However, in all cases, a default version of the command is typically invoked by other accelerator settings.

The KOKKOS package requires a "-k on" *command-line switch* respectively, which invokes a "package kokkos" command with default settings.

For the GPU, INTEL, and OPENMP packages, if a "-sf gpu" or "-sf intel" or "-sf omp" *command-line switch* is used to auto-append accelerator suffixes to various styles in the input script, then those switches also invoke a "package gpu", "package intel", or "package omp" command with default settings.

Note

A package command for a particular style can be invoked multiple times when a simulation is setup, e.g. by the *-c on, -k on, -sf, and -pk command-line switches*, and by using this command in an input script. Each time it is used all of the style options are set, either to default values or to specified settings. I.e. settings from previous invocations do not persist across multiple invocations.

See the *Accelerator packages* page for more details about using the various accelerator packages for speeding up LAMMPS simulations.

The *gpu* style invokes settings associated with the use of the GPU package.

The *Ngpu* argument sets the number of GPUs per node. If *Ngpu* is 0 and no other keywords are specified, GPU or accelerator devices are auto-selected. In this process, all platforms are searched for accelerator devices and GPUs are chosen if available. The device with the highest number of compute cores is selected. The number of devices is increased to be the number of matching accelerators with the same number of compute cores. If there are more devices than MPI tasks, the additional devices will be unused. The auto-selection of GPUs/ accelerator devices and platforms can be restricted by specifying a non-zero value for *Ngpu* and / or using the *gpuID*, *platform*, and *device_type* keywords as described below. If there are more MPI tasks (per node) than GPUs, multiple MPI tasks will share each GPU.

Optional keyword/value pairs can also be specified. Each has a default value as listed below.

The *neigh* keyword specifies where neighbor lists for pair style computation will be built. If *neigh* is *yes*, which is the default, neighbor list building is performed on the GPU. If *neigh* is *no*, neighbor list building is performed on the CPU. GPU neighbor list building currently cannot be used with a triclinic box. GPU neighbor lists are not compatible with commands that are not GPU-enabled. When a non-GPU enabled command requires a neighbor list, it will also be built on the CPU. In these cases, it will typically be more efficient to only use CPU neighbor list builds.

The *newton* keyword sets the Newton flags for pairwise (not bonded) interactions to *off* or *on*, the same as the *newton* command allows. Currently, only an *off* value is allowed, since all the GPU package pair styles require this setting. This means more computation is done, but less communication. In the future a value of *on* may be allowed, so the *newton* keyword is included as an option for compatibility with the package command for other accelerator styles. Note that the newton setting for bonded interactions is not affected by this keyword.

The *pair/only* keyword can change how any "gpu" suffix is applied. By default a suffix is applied to all styles for which an accelerated variant is available. However, that is not always the most effective way to use an accelerator. With *pair/only* set to *on* the suffix will only by applied to supported pair styles, which tend to be the most effective in using an accelerator and their operation can be overlapped with all other computations on the CPU.

The *binsize* keyword sets the size of bins used to bin atoms in neighbor list builds performed on the GPU, if *neigh* = *yes* is set. If *binsize* is set to 0.0 (the default), then the binsize is set automatically using heuristics in the GPU package.

The *split* keyword can be used for load balancing force calculations between CPU and GPU cores in GPU-enabled pair styles. If 0 < *split* < 1.0, a fixed fraction of particles is offloaded to the GPU while force calculation for the other particles occurs simultaneously on the CPU. If *split* < 0.0, the optimal fraction (based on CPU and GPU timings) is calculated every 25 timesteps, i.e. dynamic load-balancing across the CPU and GPU is performed. If *split* = 1.0, all force

calculations for GPU accelerated pair styles are performed on the GPU. In this case, other *hybrid* pair interactions, *bond*, *angle*, *dihedral*, *improper*, and *long-range* calculations can be performed on the CPU while the GPU is performing force calculations for the GPU-enabled pair style. If all CPU force computations complete before the GPU completes, LAMMPS will block until the GPU has finished before continuing the timestep.

As an example, if you have two GPUs per node and 8 CPU cores per node, and would like to run on 4 nodes (32 cores) with dynamic balancing of force calculation across CPU and GPU cores, you could specify

```
mpirun -np 32 -sf gpu -in in.script    # launch command
package gpu 2 split -1                 # input script command
```

In this case, all CPU cores and GPU devices on the nodes would be utilized. Each GPU device would be shared by 4 CPU cores. The CPU cores would perform force calculations for some fraction of the particles at the same time the GPUs performed force calculation for the other particles.

The *gpuID* keyword is used to specify the first ID for the GPU or other accelerator that LAMMPS will use. For example, if the ID is 1 and *Ngpu* is 3, GPUs 1-3 will be used. Device IDs should be determined from the output of nvc_get_devices, ocl_get_devices, or hip_get_devices as provided in the lib/gpu directory. When using OpenCL with accelerators that have main memory NUMA, the accelerators can be split into smaller virtual accelerators for more efficient use with MPI.

The *tpa* keyword sets the number of GPU vector lanes per atom used to perform force calculations. With a default value of 1, the number of lanes will be chosen based on the pair style, however, the value can be set explicitly with this keyword to fine-tune performance. For large cutoffs or with a small number of particles per GPU, increasing the value can improve performance. The number of lanes per atom must be a power of 2 and currently cannot be greater than the SIMD width for the GPU / accelerator. In the case it exceeds the SIMD width, it will automatically be decreased to meet the restriction.

The *blocksize* keyword allows you to tweak the number of threads used per thread block. This number should be a multiple of 32 (for GPUs) and its maximum depends on the specific GPU hardware. Typical choices are 64, 128, or 256. A larger block size increases occupancy of individual GPU cores, but reduces the total number of thread blocks, thus may lead to load imbalance. On modern hardware, the sensitivity to the blocksize is typically low.

The *Nthreads* value for the *omp* keyword sets the number of OpenMP threads allocated for each MPI task. This setting controls OpenMP parallelism only for routines run on the CPUs. For more details on setting the number of OpenMP threads, see the discussion of the *Nthreads* setting on this page for the "package omp" command. The meaning of *Nthreads* is exactly the same for the GPU, INTEL, and GPU packages.

The *platform* keyword is only used with OpenCL to specify the ID for an OpenCL platform. See the output from ocl_get_devices in the lib/gpu directory. In LAMMPS only one platform can be active at a time and by default (id=-1) the platform is auto-selected to find the GPU with the most compute cores. When *Ngpu* or other keywords are specified, the auto-selection is appropriately restricted. For example, if *Ngpu* is 3, only platforms with at least 3 accelerators are considered. Similar restrictions can be enforced by the *gpuID* and *device_type* keywords.

The *device_type* keyword can be used for OpenCL to specify the type of GPU to use or specify a custom configuration for an accelerator. In most cases this selection will be automatic and there is no need to use the keyword. The *applegpu* type is not specific to a particular GPU vendor, but is separate due to the more restrictive Apple OpenCL implementation. For expert users, to specify a custom configuration, the *custom* keyword followed by the next parameters can be specified:

CONFIG_ID, SIMD_SIZE, MEM_THREADS, SHUFFLE_AVAIL, FAST_MATH, THREADS_PER_ATOM, THREADS_PER_CHARGE, THREADS_PER_THREE, BLOCK_PAIR, BLOCK_BIO_PAIR, BLOCK_ELLIPSE, PPPM_BLOCK_1D, BLOCK_NBOR_BUILD, BLOCK_CELL_2D, BLOCK_CELL_ID, MAX_SHARED_TYPES, MAX_BIO_SHARED_TYPES, PPPM_MAX_SPLINE, NBOR_PREFETCH.

CONFIG_ID can be 0. SHUFFLE_AVAIL in {0,1} indicates that inline-PTX (NVIDIA) or OpenCL extensions (Intel) should be used for horizontal vector operations. FAST_MATH in {0,1} indicates that OpenCL fast math optimizations are used during the build and hardware-accelerated transcendental functions are used when available. THREADS_PER_* give the default *tpa* values for ellipsoidal models, styles using charge, and any other styles. The

BLOCK_* parameters specify the block sizes for various kernel calls and the MAX_*SHARED*_ parameters are used to determine the amount of local shared memory to use for storing model parameters.

For OpenCL, the routines are compiled at runtime for the specified GPU or accelerator architecture. The *ocl_args* keyword can be used to specify additional flags for the runtime build.

The *intel* style invokes settings associated with the use of the INTEL package. The keywords *balance*, *ghost*, *tpc*, and *tptask* are **only** applicable if LAMMPS was built with Xeon Phi co-processor support and are otherwise ignored.

The *Nphi* argument sets the number of co-processors per node. This can be set to any value, including 0, if LAMMPS was not built with co-processor support.

Optional keyword/value pairs can also be specified. Each has a default value as listed below.

The *Nthreads* value for the *omp* keyword sets the number of OpenMP threads allocated for each MPI task. This setting controls OpenMP parallelism only for routines run on the CPUs. For more details on setting the number of OpenMP threads, see the discussion of the *Nthreads* setting on this page for the "package omp" command. The meaning of *Nthreads* is exactly the same for the GPU, INTEL, and GPU packages.

The *mode* keyword determines the precision mode to use for computing pair style forces, either on the CPU or on the co-processor, when using a INTEL supported *pair style*. It can take a value of *single*, *mixed* which is the default, or *double*. *Single* means single precision is used for the entire force calculation. *Mixed* means forces between a pair of atoms are computed in single precision, but accumulated and stored in double precision, including storage of forces, torques, energies, and virial quantities. *Double* means double precision is used for the entire force calculation.

The *lrt* keyword can be used to enable "Long Range Thread (LRT)" mode. It can take a value of *yes* to enable and *no* to disable. LRT mode generates an extra thread (in addition to any OpenMP threads specified with the OMP_NUM_THREADS environment variable or the *omp* keyword). The extra thread is dedicated for performing part of the *PPPM solver* computations and communications. This can improve parallel performance on processors supporting Simultaneous Multithreading (SMT) such as Hyper-Threading (HT) on Intel processors. In this mode, one additional thread is generated per MPI process. LAMMPS will generate a warning in the case that more threads are used than available in SMT hardware on a node. If the PPPM solver from the INTEL package is not used, then the LRT setting is ignored and no extra threads are generated. Enabling LRT will replace the *run_style* with the *verlet/lrt/intel* style that is identical to the default *verlet* style aside from supporting the LRT feature. This feature requires setting the pre-processor flag -DLMP_INTEL_USELRT in the makefile when compiling LAMMPS.

The *balance* keyword sets the fraction of *pair style* work offloaded to the co-processor for split values between 0.0 and 1.0 inclusive. While this fraction of work is running on the co-processor, other calculations will run on the host, including neighbor and pair calculations that are not offloaded, as well as angle, bond, dihedral, kspace, and some MPI communications. If *split* is set to -1, the fraction of work is dynamically adjusted automatically throughout the run. This typically give performance within 5 to 10 percent of the optimal fixed fraction.

The *ghost* keyword determines whether or not ghost atoms, i.e. atoms at the boundaries of processor subdomains, are offloaded for neighbor and force calculations. When the value = "no", ghost atoms are not offloaded. This option can reduce the amount of data transfer with the co-processor and can also overlap MPI communication of forces with computation on the co-processor when the *newton pair* setting is "on". When the value = "yes", ghost atoms are offloaded. In some cases this can provide better performance, especially if the *balance* fraction is high.

The *tpc* keyword sets the max # of co-processor threads *Ntpc* that will run on each core of the co-processor. The default value = 4, which is the number of hardware threads per core supported by the current generation Xeon Phi chips.

The *tptask* keyword sets the max # of co-processor threads (Ntptask* assigned to each MPI task. The default value = 240, which is the total # of threads an entire current generation Xeon Phi chip can run (240 = 60 cores * 4 threads/core). This means each MPI task assigned to the Phi will enough threads for the chip to run the max allowed, even if only 1 MPI task is assigned. If 8 MPI tasks are assigned to the Phi, each will run with 30 threads. If you wish to limit the number of threads per MPI task, set *tptask* to a smaller value. E.g. for *tptask* = 16, if 8 MPI tasks are assigned, each will run with 16 threads, for a total of 128.

Note that the default settings for *tpc* and *tptask* are fine for most problems, regardless of how many MPI tasks you assign to a Phi.

Added in version 15Jun2023.

The *pppm_table* keyword with the argument yes allows to use a pre-computed table to efficiently spread the charge to the PPPM grid. This feature is enabled by default but can be turned off using the keyword with the argument *no*.

The *no_affinity* keyword will turn off automatic setting of core affinity for MPI tasks and OpenMP threads on the host when using offload to a co-processor. Affinity settings are used when possible to prevent MPI tasks and OpenMP threads from being on separate NUMA domains and to prevent offload threads from interfering with other processes/threads used for LAMMPS.

The *kokkos* style invokes settings associated with the use of the KOKKOS package.

All of the settings are optional keyword/value pairs. Each has a default value as listed below.

The *neigh* keyword determines how neighbor lists are built. A value of *half* uses a thread-safe variant of half-neighbor lists, the same as used by most pair styles in LAMMPS, which is the default when running on CPUs (i.e. the Kokkos CUDA back end is not enabled).

A value of *full* uses a full neighbor lists and is the default when running on GPUs. This performs twice as much computation as the *half* option, however that is often a win because it is thread-safe and does not require atomic operations in the calculation of pair forces. For that reason, *full* is the default setting for GPUs. However, when running on CPUs, a *half* neighbor list is the default because it are often faster, just as it is for non-accelerated pair styles. Similarly, the *neigh/qeq* keyword determines how neighbor lists are built for *fix qeq/reaxff/kk*.

If the *neigh/thread* keyword is set to *off*, then the KOKKOS package threads only over atoms. However, for small systems, this may not expose enough parallelism to keep a GPU busy. When this keyword is set to *on*, the KOKKOS package threads over both atoms and neighbors of atoms. When using *neigh/thread on*, the *newton pair* setting must be "off". Using *neigh/thread on* may be slower for large systems, so this this option is turned on by default only when running on one or more GPUs and there are 16k atoms or less owned by an MPI rank. Not all KOKKOS-enabled potentials support this keyword yet, and only thread over atoms. Many simple pairwise potentials such as Lennard-Jones do support threading over both atoms and neighbors.

If the *neigh/transpose* keyword is set to *off*, then the KOKKOS package will use the same memory layout for building the neighbor list on GPUs as used for the pair style. When this keyword is set to *on* it will use a different (transposed) memory layout to build the neighbor list on GPUs. This can be faster in some cases (e.g. ReaxFF HNS benchmark) but slower in others (e.g. Lennard Jones benchmark). The copy between different memory layouts is done out of place and therefore doubles the memory overhead of the neighbor list, which can be significant.

The *newton* keyword sets the Newton flags for pairwise and bonded interactions to *off* or *on*, the same as the *newton* command allows. The default for GPUs is *off* because this will almost always give better performance for the KOKKOS package. This means more computation is done, but less communication. However, when running on CPUs a value of *on* is the default since it can often be faster, just as it is for non-accelerated pair styles

The *binsize* keyword sets the size of bins used to bin atoms during neighbor list builds. The same value can be set by the *neigh_modify binsize* command. Making it an option in the package kokkos command allows it to be set from the command line. The default value for CPUs is 0.0, which means the LAMMPS default will be used, which is bins = 1/2 the size of the pairwise cutoff + neighbor skin distance. This is fine when neighbor lists are built on the CPU. For GPU builds, a 2x larger binsize equal to the pairwise cutoff + neighbor skin is often faster, which is the default. Note that if you use a longer-than-usual pairwise cutoff, e.g. to allow for a smaller fraction of KSpace work with a *long-range Coulombic solver* because the GPU is faster at performing pairwise interactions, then this rule of thumb may give too large a binsize and the default should be overridden with a smaller value.

The *comm* and *comm/exchange* and *comm/forward* and *comm/pair/forward* and *comm/fix/forward* and *comm/reverse* and *comm/pair/reverse* keywords determine whether the host or device performs the packing and unpacking of data when communicating per-atom data between processors. "Exchange" communication happens only on timesteps that

neighbor lists are rebuilt. The data is only for atoms that migrate to new processors. "Forward" communication happens every timestep. "Reverse" communication happens every timestep if the *newton* option is on. The data is for atom coordinates and any other atom properties that needs to be updated for ghost atoms owned by each processor. "Pair/comm" controls additional communication in pair styles, such as pair_style EAM. "Fix/comm" controls additional communication in fixes, such as fix SHAKE.

The *comm* keyword is simply a short-cut to set the same value for all the comm keywords.

The value options for the keywords are *no* or *host* or *device*. A value of *no* means to use the standard non-KOKKOS method of packing/unpacking data for the communication. A value of *host* means to use the host, typically a multicore CPU, and perform the packing/unpacking in parallel with threads. A value of *device* means to use the device, typically a GPU, to perform the packing/unpacking operation.

For the *comm/pair/forward* or *comm/fix/forward* or *comm/pair/reverse* keywords, if a value of *host* is used it will be automatically be changed to *no* since these keywords don't support *host* mode. The value of *no* will also always be used when running on the CPU, i.e. setting the value to *device* will have no effect if the pair/fix style is running on the CPU. For the *comm/fix/forward* or *comm/pair/reverse* keywords, not all styles support *device* mode and in that case will run in *no* mode instead.

The optimal choice for these keywords depends on the input script and the hardware used. The *no* value is useful for verifying that the Kokkos-based *host* and *device* values are working correctly. It is the default when running on CPUs since it is usually the fastest.

When running on CPUs or Xeon Phi, the *host* and *device* values work identically. When using GPUs, the *device* value is the default since it will typically be optimal if all of your styles used in your input script are supported by the KOKKOS package. In this case data can stay on the GPU for many timesteps without being moved between the host and GPU, if you use the *device* value. If your script uses styles (e.g. fixes) which are not yet supported by the KOKKOS package, then data has to be moved between the host and device anyway, so it is typically faster to let the host handle communication, by using the *host* value. Using *host* instead of *no* will enable use of multiple threads to pack/unpack communicated data. When running small systems on a GPU, performing the exchange pack/unpack on the host CPU can give speedup since it reduces the number of CUDA kernel launches.

The *sort* keyword determines whether the host or device performs atom sorting, see the *atom_modify sort* command. The value options for the *sort* keyword are *no* or *device* similar to the *comm* keywords above. If a value of *host* is used it will be automatically be changed to *no* since the *sort* keyword does not support *host* mode. Not all fix styles with extra atom data support *device* mode and in that case a warning will be given and atom sorting will run in *no* mode instead.

Added in version 17Apr2024.

The *atom/map* keyword determines whether the host or device builds the atom_map, see the *atom_modify map* command. The value options for the *atom/map* keyword are identical to the *sort* keyword above.

The *gpu/aware* keyword chooses whether GPU-aware MPI will be used. When this keyword is set to *on*, buffers in GPU memory are passed directly through MPI send/receive calls. This reduces overhead of first copying the data to the host CPU. However GPU-aware MPI is not supported on all systems, which can lead to segmentation faults and would require using a value of *off*. If LAMMPS can safely detect that GPU-aware MPI is not available (currently only possible with OpenMPI v2.0.0 or later), then the *gpu/aware* keyword is automatically set to *off* by default. When the *gpu/aware* keyword is set to *off* while any of the *comm* keywords are set to *device*, the value for these *comm* keywords will be automatically changed to *no*. This setting has no effect if not running on GPUs or if using only one MPI rank. GPU-aware MPI is available for OpenMPI 1.8 (or later versions), Mvapich2 1.9 (or later) when the "MV2_USE_CUDA" environment variable is set to "1", CrayMPI, and IBM Spectrum MPI when the "-gpu" flag is used.

The *pair/only* keyword can change how the KOKKOS suffix "kk" is applied when using an accelerator device. By default device acceleration is always used for all available styles. With *pair/only* set to *on* the suffix setting will choose device acceleration only for pair styles and run all other force computations on the host CPU. The *comm* flags, along with the *sort* and *atom/map* keywords will also automatically be changed to *no*. This can result in better performance for certain configurations and system sizes.

The *omp* style invokes settings associated with the use of the OPENMP package.

The *Nthreads* argument sets the number of OpenMP threads allocated for each MPI task. For example, if your system has nodes with dual quad-core processors, it has a total of 8 cores per node. You could use two MPI tasks per node (e.g. using the -ppn option of the mpirun command in MPICH or -npernode in OpenMPI), and set *Nthreads* = 4. This would use all 8 cores on each node. Note that the product of MPI tasks * threads/task should not exceed the physical number of cores (on a node), otherwise performance will suffer.

Setting *Nthreads* = 0 instructs LAMMPS to use whatever value is the default for the given OpenMP environment. This is usually determined via the *OMP_NUM_THREADS* environment variable or the compiler runtime. Note that in most cases the default for OpenMP capable compilers is to use one thread for each available CPU core when *OMP_NUM_THREADS* is not explicitly set, which can lead to poor performance.

Here are examples of how to set the environment variable when launching LAMMPS:

```
env OMP_NUM_THREADS=4 lmp_machine -sf omp -in in.script
env OMP_NUM_THREADS=2 mpirun -np 2 lmp_machine -sf omp -in in.script
mpirun -x OMP_NUM_THREADS=2 -np 2 lmp_machine -sf omp -in in.script
```

or you can set it permanently in your shell's start-up script. All three of these examples use a total of 4 CPU cores.

Note that different MPI implementations have different ways of passing the OMP_NUM_THREADS environment variable to all MPI processes. The second example line above is for MPICH; the third example line with -x is for OpenMPI. Check your MPI documentation for additional details.

What combination of threads and MPI tasks gives the best performance is difficult to predict and can depend on many components of your input. Not all features of LAMMPS support OpenMP threading via the OPENMP package and the parallel efficiency can be very different, too.

Note

If you build LAMMPS with the GPU, INTEL, and / or OPENMP packages, be aware these packages all allow setting of the *Nthreads* value via their package commands, but there is only a single global *Nthreads* value used by OpenMP. Thus if multiple package commands are invoked, you should ensure the values are consistent. If they are not, the last one invoked will take precedence, for all packages. Also note that if the *-sf hybrid intel omp command-line switch* is used, it invokes a "package intel" command, followed by a "package omp" command, both with a setting of *Nthreads* = 0. Likewise for a hybrid suffix for gpu and omp. Note that KOKKOS also supports setting the number of OpenMP threads from the command line using the "-k on" *command-line switch*. The default for KOKKOS is 1 thread per MPI task, so any other number of threads should be explicitly set using the "-k on" command-line switch (and this setting should be consistent with settings from any other packages used).

Optional keyword/value pairs can also be specified. Each has a default value as listed below.

The *neigh* keyword specifies whether neighbor list building will be multi-threaded in addition to force calculations. If *neigh* is set to *no* then neighbor list calculation is performed only by MPI tasks with no OpenMP threading. If *mode* is *yes* (the default), a multi-threaded neighbor list build is used. Using *neigh* = *yes* is almost always faster and should produce identical neighbor lists at the expense of using more memory. Specifically, neighbor list pages are allocated for all threads at the same time and each thread works within its own pages.

1.73.4 Restrictions

This command cannot be used after the simulation box is defined by a *read_data* or *create_box* command.

The *gpu* style of this command can only be invoked if LAMMPS was built with the GPU package. See the *Build package* doc page for more info.

The *intel* style of this command can only be invoked if LAMMPS was built with the INTEL package. See the *Build package* page for more info.

The *kokkos* style of this command can only be invoked if LAMMPS was built with the KOKKOS package. See the *Build package* doc page for more info.

The *omp* style of this command can only be invoked if LAMMPS was built with the OPENMP package. See the *Build package* doc page for more info.

1.73.5 Related commands

suffix, *-pk command-line switch*

1.73.6 Defaults

For the GPU package, the default parameters and settings are:

```
Ngpu = 0, neigh = yes, newton = off, binsize = 0.0, split = 1.0, gpuID = 0 to Ngpu-1, tpa = 1, omp = 0,
↪ platform=-1.
```

These settings are made automatically if the "-sf gpu" *command-line switch* is used. If it is not used, you must invoke the package gpu command in your input script or via the "-pk gpu" *command-line switch*.

For the INTEL package, the default parameters and settings are:

```
Nphi = 1, omp = 0, mode = mixed, lrt = no, balance = -1, tpc = 4, tptask = 240, pppm_table = yes
```

The default ghost option is determined by the pair style being used. This value is output to the screen in the offload report at the end of each run. Note that all of these settings, except "omp" and "mode", are ignored if LAMMPS was not built with Xeon Phi co-processor support. These settings are made automatically if the "-sf intel" *command-line switch* is used. If it is not used, you must invoke the package intel command in your input script or via the "-pk intel" *command-line switch*.

For the KOKKOS package when using GPUs, the option defaults are:

```
neigh = full, neigh/qeq = full, newton = off, binsize = 2x LAMMPS default value, comm = device, sort␣
↪= device, atom/map = device, neigh/transpose = off, gpu/aware = on
```

For GPUs, option neigh/thread = on when there are 16k atoms or less on an MPI rank, otherwise it is "off". When LAMMPS can safely detect that GPU-aware MPI is not available, the default value of gpu/aware becomes "off".

For the KOKKOS package when using CPUs or Xeon Phis, the option defaults are:

```
neigh = half, neigh/qeq = half, newton = on, binsize = 0.0, comm = no, sort = no, atom/map = no
```

These settings are made automatically by the required "-k on" *command-line switch*. You can change them by using the package kokkos command in your input script or via the *-pk kokkos command-line switch*.

For the OMP package, the defaults are

```
Nthreads = 0, neigh = yes
```

These settings are made automatically if the "-sf omp" *command-line switch* is used. If it is not used, you must invoke the package omp command in your input script or via the "-pk omp" *command-line switch*.

1.74 pair_coeff command

1.74.1 Syntax

```
pair_coeff I J args
```

- I,J = numeric atom types (see asterisk form below), or type labels
- args = coefficients for one or more pairs of atom types

1.74.2 Examples

```
pair_coeff 1 2 1.0 1.0 2.5
pair_coeff 2 * 1.0 1.0
pair_coeff 3* 1*2 1.0 1.0 2.5
pair_coeff * * 1.0 1.0
pair_coeff * * nialhjea 1 1 2
pair_coeff * 3 morse.table ENTRY1
pair_coeff 1 2 lj/cut 1.0 1.0 2.5 # (for pair_style hybrid)

labelmap atom 1 C
labelmap atom 2 H
pair_coeff C H 1.0 1.0 2.5
```

1.74.3 Description

Specify the pairwise force field coefficients for one or more pairs of atom types. The number and meaning of the coefficients depends on the pair style. Pair coefficients can also be set in the data file read by the *read_data* command or in a restart file.

I and J can be specified in one of several ways. Explicit numeric values can be used for each, as in the first example above. Or, one or both of the types in the I,J pair can be a type label, which is an alphanumeric string defined by the *labelmap* command or in a section of a data file read by the *read_data* command, and which converts internally to a numeric type. Internally, LAMMPS will set coefficients for the symmetric J,I interaction to the same values as the I,J interaction.

For numeric values only, a wildcard asterisk can be used in place of or in conjunction with the I,J arguments to set the coefficients for multiple pairs of atom types. This takes the form "*" or "*n" or "n*" or "m*n". If *N* is the number of atom types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to *N* (inclusive). A middle asterisk means all types from m to n (inclusive). For the asterisk syntax, only type pairs with I <= J are considered; if asterisks imply type pairs where J < I, they are ignored. Again internally, LAMMPS will set the coefficients for the symmetric J,I interactions to the same values as the I <= J interactions.

Note that a pair_coeff command can override a previous setting for the same I,J pair. For example, these commands set the coeffs for all I,J pairs, then overwrite the coeffs for just the I,J = 2,3 pair:

```
pair_coeff * * 1.0 1.0 2.5
pair_coeff 2 3 2.0 1.0 1.12
```

A line in a data file that specifies pair coefficients uses the exact same format as the arguments of the pair_coeff command in an input script, with the exception of the I,J type arguments. In each line of the "Pair Coeffs" section of a data file, only a single type I is specified, which sets the coefficients for type I interacting with type I. This is because the section has exactly *N* lines, where *N* is the number of atom types. For this reason, the wild-card asterisk should also not be used as part of the I argument. Thus in a data file, the line corresponding to the first example above would be listed as

```
2 1.0 1.0 2.5
```

For many potentials, if coefficients for type pairs with I != J are not set explicitly by a pair_coeff command, the values are inferred from the I,I and J,J settings by mixing rules; see the *pair_modify* command for a discussion. Details on this option as it pertains to individual potentials are described on the page for the potential.

Many pair styles, typically for many-body potentials, use tabulated potential files as input, when specifying the pair_coeff command. Potential files provided with LAMMPS are in the potentials directory of the distribution. For some potentials, such as EAM, other archives of suitable files can be found on the Web. They can be used with LAMMPS so long as they are in the format LAMMPS expects, as discussed on the individual doc pages. The first line of potential files may contain metadata with upper case tags followed their value. These may be parsed and used by LAMMPS. Currently supported are the "DATE:" tag and the UNITS: tag. For pair styles that have been programmed to support the metadata, the value of the "DATE:" tag is printed to the screen and logfile so that the version of a potential file can be later identified. The UNITS: tag indicates the *units* setting required for this particular potential file. If the potential file was created for a different sets of units, LAMMPS will terminate with an error. If the potential file does not contain the tag, no check will be made and it is the responsibility of the user to determine that the unit style is correct.

In some select cases and for specific combinations of unit styles, LAMMPS is capable of automatically converting potential parameters from a file. In those cases, a warning message signaling that an automatic conversion has happened is printed to the screen.

When a pair_coeff command using a potential file is specified, LAMMPS looks for the potential file in 2 places. First it looks in the location specified. E.g. if the file is specified as "niu3.eam", it is looked for in the current working directory. If it is specified as "../potentials/niu3.eam", then it is looked for in the potentials directory, assuming it is a sister directory of the current working directory. If the file is not found, it is then looked for in one of the directories specified by the LAMMPS_POTENTIALS environment variable. Thus if this is set to the potentials directory in the LAMMPS distribution, then you can use those files from anywhere on your system, without copying them into your working directory. Environment variables are set in different ways for different shells. Here are example settings for

csh, tcsh:

```
setenv LAMMPS_POTENTIALS /path/to/lammps/potentials
```

bash:

```
export LAMMPS_POTENTIALS=/path/to/lammps/potentials
```

Windows:

```
set LAMMPS_POTENTIALS="C:\\Path to LAMMPS\\Potentials"
```

The LAMMPS_POTENTIALS environment variable may contain paths to multiple folders, if they are separated by ";" on Windows and ":" on all other operating systems, just like the PATH and similar environment variables.

The alphabetic list of pair styles defined in LAMMPS is given on the *pair_style* doc page. They are also listed in more compact form on the *Commands pair* doc page.

Click on the style to display the formula it computes and its coefficients as specified by the associated pair_coeff command.

1.74.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

1.74.5 Related commands

pair_style, *pair_modify*, *read_data*, *read_restart*, *pair_write*

1.74.6 Default

none

1.75 pair_modify command

1.75.1 Syntax

```
pair_modify keyword values ...
```

- one or more keyword/value pairs may be listed
- keyword = *pair* or *shift* or *mix* or *table* or *table/disp* or *tabinner* or *tabinner/disp* or *tail* or *compute* or *nofdotr* or *special* or *compute/tally* or *neigh/trim*

  pair value = sub-style N
    sub-style = sub-style of pair hybrid
    N = which instance of sub-style (1 to M), only specify if sub-style is used multiple times
  mix value = geometric or arithmetic or sixthpower
  shift value = yes or no
  table value = N
    2^N = # of values in table
  table/disp value = N
    2^N = # of values in table
  tabinner value = cutoff
    cutoff = inner cutoff at which to begin table (distance units)
  tabinner/disp value = cutoff
    cutoff = inner cutoff at which to begin table (distance units)
  tail value = yes or no
  compute value = yes or no
  nofdotr value = none
  special values = which wt1 wt2 wt3
    which = lj/coul or lj or coul
    w1,w2,w3 = 1-2, 1-3, 1-4 weights from 0.0 to 1.0 inclusive
  compute/tally value = yes or no
  neigh/trim value = yes or no

1.75.2 Examples

```
pair_modify shift yes mix geometric
pair_modify tail yes
pair_modify table 12
pair_modify pair lj/cut compute no
pair_modify pair tersoff compute/tally no
pair_modify pair lj/cut/coul/long 1 special lj/coul 0.0 0.0 0.0
pair_modify pair lj/cut/coul/long special lj 0.0 0.0 0.5 special coul 0.0 0.0 0.8333333
```

1.75.3 Description

Modify the parameters of the currently defined pair style. If the pair style is *hybrid or hybrid/overlay*, then the specified parameters are by default modified for all the hybrid sub-styles.

Note

The behavior for hybrid pair styles can be changed by using the *pair* keyword, which allows selection of a specific sub-style to apply all remaining keywords to. The *special* and *compute/tally* keywords can **only** be used in conjunction with the *pair* keyword. See further details about these 3 keywords below.

The *mix* keyword affects pair coefficients for interactions between atoms of type I and J, when I != J and the coefficients are not explicitly set in the input script. Note that coefficients for I = J must be set explicitly, either in the input script via the *pair_coeff* command or in the "Pair Coeffs" or "PairIJ Coeffs" sections of the *data file*. For some pair styles it is not necessary to specify coefficients when I != J, since a "mixing" rule will create them from the I,I and J,J settings. The pair_modify *mix* value determines what formulas are used to compute the mixed coefficients. In each case, the cutoff distance is mixed the same way as sigma.

Note that not all pair styles support mixing and some mix options are not available for certain pair styles. Also, there are additional restrictions when using *pair style hybrid or hybrid/overlay*. See the page for individual pair styles for those restrictions. Note also that the *pair_coeff* command also can be used to directly set coefficients for a specific I != J pairing, in which case no mixing is performed. If possible, LAMMPS will print an informational message about how many of the mixed pair coefficients were generated and which mixing rule was applied.

- mix *geometric*

$$\varepsilon_{ij} = \sqrt{\varepsilon_i \varepsilon_j}$$
$$\sigma_{ij} = \sqrt{\sigma_i \sigma_j}$$

- mix *arithmetic*

$$\varepsilon_{ij} = \sqrt{\varepsilon_i \varepsilon_j}$$
$$\sigma_{ij} = \frac{1}{2}(\sigma_i + \sigma_j)$$

- mix *sixthpower*

$$\varepsilon_{ij} = \frac{2\sqrt{\varepsilon_i \varepsilon_j}\sigma_i^3 \sigma_j^3}{\sigma_i^6 + \sigma_j^6}$$
$$\sigma_{ij} = \left(\frac{1}{2}(\sigma_i^6 + \sigma_j^6)\right)^{\frac{1}{6}}$$

The *shift* keyword determines whether a Lennard-Jones potential is shifted at its cutoff to 0.0. If so, this adds an energy term to each pairwise interaction which will be included in the thermodynamic output, but does not affect pair forces or atom trajectories. See the doc page for individual pair styles to see which ones support this option.

The *table* and *table/disp* keywords apply to pair styles with a long-range Coulombic term or long-range dispersion term respectively; see the page for individual styles to see which potentials support these options. If N is non-zero, a table of length 2^N is pre-computed for forces and energies, which can shrink their computational cost by up to a factor of 2. The table is indexed via a bit-mapping technique *(Wolff)* and a linear interpolation is performed between adjacent table values. In our experiments with different table styles (lookup, linear, spline), this method typically gave the best performance in terms of speed and accuracy.

The choice of table length is a tradeoff in accuracy versus speed. A larger N yields more accurate force computations, but requires more memory which can slow down the computation due to cache misses. A reasonable value of N is between 8 and 16. The default value of 12 (table of length 4096) gives approximately the same accuracy as the no-table (N = 0) option. For N = 0, forces and energies are computed directly, using a polynomial fit for the needed erfc() function evaluation, which is what earlier versions of LAMMPS did. Values greater than 16 typically slow down the simulation and will not improve accuracy; values from 1 to 8 give unreliable results.

The *tabinner* and *tabinner/disp* keywords set an inner cutoff above which the pairwise computation is done by table lookup (if tables are invoked), for the corresponding Coulombic and dispersion tables discussed with the *table* and *table/disp* keywords. The smaller the cutoff is set, the less accurate the table becomes (for a given number of table values), which can require use of larger tables. The default cutoff value is sqrt(2.0) distance units which means nearly all pairwise interactions are computed via table lookup for simulations with "real" units, but some close pairs may be computed directly (non-table) for simulations with "lj" units.

When the *tail* keyword is set to *yes*, certain pair styles will add a long-range VanderWaals tail "correction" to the energy and pressure. These corrections are bookkeeping terms which do not affect dynamics, unless a constant-pressure simulation is being performed. See the page for individual styles to see which support this option. These corrections are included in the calculation and printing of thermodynamic quantities (see the *thermo_style* command). Their effect will also be included in constant NPT or NPH simulations where the pressure influences the simulation box dimensions (e.g. the *fix npt* and *fix nph* commands). The formulas used for the long-range corrections come from equation 5 of *(Sun)*.

Note

The tail correction terms are computed at the beginning of each run, using the current atom counts of each atom type. If atoms are deleted (or lost) or created during a simulation, e.g. via the *fix gcmc* command, the correction factors are not re-computed. If you expect the counts to change dramatically, you can break a run into a series of shorter runs so that the correction factors are re-computed more frequently.

Several additional assumptions are inherent in using tail corrections, including the following:

- The simulated system is a 3d bulk homogeneous liquid. This option should not be used for systems that are non-liquid, 2d, have a slab geometry (only 2d periodic), or inhomogeneous.
- G(r), the radial distribution function (rdf), is unity beyond the cutoff, so a fairly large cutoff should be used (i.e. 2.5 sigma for an LJ fluid), and it is probably a good idea to verify this assumption by checking the rdf. The rdf is not exactly unity beyond the cutoff for each pair of interaction types, so the tail correction is necessarily an approximation.

  The tail corrections are computed at the beginning of each simulation run. If the number of atoms changes during the run, e.g. due to atoms leaving the simulation domain, or use of the *fix gcmc* command, then the corrections are not updated to reflect the changed atom count. If this is a large effect in your simulation, you should break the long run into several short runs, so that the correction factors are re-computed multiple times.
- Thermophysical properties obtained from calculations with this option enabled will not be thermodynamically consistent with the truncated force-field that was used. In other words, atoms do not feel any LJ pair interactions

beyond the cutoff, but the energy and pressure reported by the simulation include an estimated contribution from those interactions.

The *compute* keyword allows pairwise computations to be turned off, even though a *pair_style* is defined. This is not useful for running a real simulation, but can be useful for debugging purposes or for performing a *rerun* simulation, when you only wish to compute partial forces that do not include the pairwise contribution.

Two examples are as follows. First, this option allows you to perform a simulation with *pair_style hybrid* with only a subset of the hybrid sub-styles enabled. Second, this option allows you to perform a simulation with only long-range interactions but no short-range pairwise interactions. Doing this by simply not defining a pair style will not work, because the *kspace_style* command requires a Kspace-compatible pair style be defined.

The *nofdotr* keyword allows to disable an optimization that computes the global stress tensor from the total forces and atom positions rather than from summing forces between individual pairs of atoms.

The *pair* keyword can only be used with the *hybrid and hybrid/overlay* pair styles. If used, it must appear first in the list of keywords.

Its meaning is that all the following parameters will only be modified for the specified sub-style. If the sub-style is defined multiple times, then an additional numeric argument *N* must also be specified, which is a number from 1 to M where M is the number of times the sub-style was listed in the *pair_style hybrid* command. The extra number indicates which instance of the sub-style the remaining keywords will be applied to.

The *special* and *compute/tally* keywords can **only** be used in conjunction with the *pair* keyword and they must directly follow it. I.e. any other keyword, must appear after *pair*, *special*, and *compute/tally*.

The *special* keyword overrides the global *special_bonds* 1-2, 1-3, 1-4 exclusion settings (weights) for the sub-style selected by the *pair* keyword.

Similar to the *special_bonds* command, it takes 4 arguments. The *which* argument can be *lj* to change only the non-Coulomb weights (e.g. Lennard-Jones or Buckingham), *coul* to change only the Coulombic settings, or *lj/coul* to change both to the same values. The *wt1,wt2,wt3* values are numeric weights from 0.0 to 1.0 inclusive, for the 1-2, 1-3, and 1-4 bond topology neighbors, respectively. The *special* keyword can be used multiple times, e.g. to set the *lj* and *coul* settings to different values.

Note

The *special* keyword is not compatible with pair styles from the GPU or the INTEL package and attempting to use it will cause an error.

Note

Weights of exactly 0.0 or 1.0 in the *special_bonds* command have implications on the neighbor list construction, which means that they cannot be overridden by using the *special* keyword. One workaround for this restriction is to use the *special_bonds* command with weights like 1.0e-10 or 0.999999999 instead of 0.0 or 1.0, respectively, which enables to reset each them to any value between 0.0 and 1.0 inclusively. Otherwise you can set **all** global weights to an arbitrary number between 0.0 or 1.0, like 0.5, and then you have to override **all** *special* settings for **all** sub-styles which use the 1-2, 1-3, and 1-4 exclusion weights in their force/energy computation.

The *compute/tally* keyword disables or enables registering *compute */tally* computes for the sub-style specified by the *pair* keyword. Use *no* to disable, or *yes* to enable.

> **Note**
>
> The "pair_modify pair compute/tally" command must be issued **before** the corresponding compute style is defined.

Added in version 3Aug2022.

The *neigh/trim* keyword controls whether an explicit cutoff is set for each neighbor list request issued by individual pair sub-styles when using *pair hybrid/overlay*. When this keyword is set to *no*, then the cutoff of each pair sub-style neighbor list will be set equal to the largest cutoff, even if a shorter cutoff is specified for a particular sub-style. If possible the neighbor list will be copied directly from another list. When this keyword is set to *yes* then the cutoff of the neighbor list will be explicitly set to the value requested by the pair sub-style, and if possible the list will be created by trimming neighbors from another list with a longer cutoff, otherwise a new neighbor list will be created with the specified cutoff. The *yes* option can be faster when there are multiple pair styles with different cutoffs since the number of pair-wise distance checks between neighbors is reduced (but the time required to build the neighbor lists is increased). The *no* option could be faster when two or more neighbor lists have similar (but not exactly the same) cutoffs.

> **Note**
>
> The "pair_modify neigh/trim" command *only* applies when there are multiple pair sub-styles for the same atoms with different cutoffs, i.e. when using pair style hybrid/overlay. If you have different cutoffs for different pairs for atoms type, the *neighbor style multi* should be used to create optimized neighbor lists.

1.75.4 Restrictions

You cannot use *shift* yes with *tail* yes, since those are conflicting options. You cannot use *tail* yes with 2d simulations. You cannot use *special* with pair styles from the GPU or INTEL package.

1.75.5 Related commands

pair_style, *pair_style hybrid*, *pair_coeff*, *thermo_style*, *compute */tally*, *neighbor multi*

1.75.6 Default

The option defaults are mix = geometric, shift = no, table = 12, tabinner = sqrt(2.0), tail = no, compute = yes, and neigh/trim yes.

Note that some pair styles perform mixing, but only a certain style of mixing. See the doc pages for individual pair styles for details.

(Wolff) Wolff and Rudd, Comp Phys Comm, 120, 200-32 (1999).

(Sun) Sun, J Phys Chem B, 102, 7338-7364 (1998).

1.76 pair_style command

1.76.1 Syntax

```
pair_style style args
```

- style = one of the styles from the list below
- args = arguments used by a particular style

1.76.2 Examples

```
pair_style lj/cut 2.5
pair_style eam/alloy
pair_style hybrid lj/charmm/coul/long 10.0 eam
pair_style table linear 1000
pair_style none
```

1.76.3 Description

Set the formula(s) LAMMPS uses to compute pairwise interactions. In LAMMPS, pair potentials are defined between pairs of atoms that are within a cutoff distance and the set of active interactions typically changes over time. See the *bond_style* command to define potentials between pairs of bonded atoms, which typically remain in place for the duration of a simulation.

In LAMMPS, pairwise force fields encompass a variety of interactions, some of which include many-body effects, e.g. EAM, Stillinger-Weber, Tersoff, REBO potentials. They are still classified as "pairwise" potentials because the set of interacting atoms changes with time (unlike molecular bonds) and thus a neighbor list is used to find nearby interacting atoms.

Hybrid models where specified pairs of atom types interact via different pair potentials can be setup using the *hybrid* pair style.

The coefficients associated with a pair style are typically set for each pair of atom types, and are specified by the *pair_coeff* command or read from a file by the *read_data* or *read_restart* commands.

The *pair_modify* command sets options for mixing of type I-J interaction coefficients and adding energy offsets or tail corrections to Lennard-Jones potentials. Details on these options as they pertain to individual potentials are described on the doc page for the potential. Likewise, info on whether the potential information is stored in a *restart file* is listed on the potential doc page.

In the formulas listed for each pair style, E is the energy of a pairwise interaction between two atoms separated by a distance r. The force between the atoms is the negative derivative of this expression.

If the pair_style command has a cutoff argument, it sets global cutoffs for all pairs of atom types. The distance(s) can be smaller or larger than the dimensions of the simulation box.

In many cases, the global cutoff value can be overridden for a specific pair of atom types by the *pair_coeff* command.

If a new pair_style command is specified with a new style, all previous *pair_coeff* and *pair_modify* command settings are erased; those commands must be re-specified if necessary.

If a new pair_style command is specified with the same style, then only the global settings in that command are reset. Any previous doc:*pair_coeff* <*pair_coeff*> and *pair_modify* command settings are preserved. The only exception is

that if the global cutoff in the pair_style command is changed, it will override the corresponding cutoff in any of the previous *pair_modify* commands.

Two pair styles which do not follow this rule are the pair_style *table* and *hybrid* commands. A new pair_style command for these styles will wipe out all previously specified *pair_coeff* and *pair_modify* settings, including for the sub-styles of the *hybrid* command.

Here is an alphabetic list of pair styles defined in LAMMPS. They are also listed in more compact form on the *Commands pair* doc page.

Click on the style to display the formula it computes, any additional arguments specified in the pair_style command, and coefficients specified by the associated *pair_coeff* command.

There are also additional accelerated pair styles included in the LAMMPS distribution for faster performance on CPUs, GPUs, and KNLs. The individual style names on the *Commands pair* doc page are followed by one or more of (g,i,k,o,t) to indicate which accelerated styles exist.

- *none* - turn off pairwise interactions
- *hybrid* - multiple styles of pairwise interactions
- *hybrid/molecular* - different pair styles for intra- and inter-molecular interactions
- *hybrid/overlay* - multiple styles of superposed pairwise interactions
- *hybrid/scaled* - multiple styles of scaled superposed pairwise interactions
- *zero* - neighbor list but no interactions
- *adp* - angular dependent potential (ADP) of Mishin
- *agni* - AGNI machine-learning potential
- *aip/water/2dm* - anisotropic interfacial potential for water in 2d geometries
- *airebo* - AIREBO potential of Stuart
- *airebo/morse* - AIREBO with Morse instead of LJ
- *amoeba* -
- *atm* - Axilrod-Teller-Muto potential
- *awpmd/cut* - Antisymmetrized Wave Packet MD potential for atoms and electrons
- *beck* - Beck potential
- *body/nparticle* - interactions between body particles
- *body/rounded/polygon* - granular-style 2d polygon potential
- *body/rounded/polyhedron* - granular-style 3d polyhedron potential
- *bop* - BOP potential of Pettifor
- *born* - Born-Mayer-Huggins potential
- *born/coul/dsf* - Born with damped-shifted-force model
- *born/coul/dsf/cs* - Born with damped-shifted-force and core/shell model
- *born/coul/long* - Born with long-range Coulomb
- *born/coul/long/cs* - Born with long-range Coulomb and core/shell
- *born/coul/msm* - Born with long-range MSM Coulomb

- *born/coul/wolf* - Born with Wolf potential for Coulomb
- *born/coul/wolf/cs* - Born with Wolf potential for Coulomb and core/shell model
- *born/gauss* - Born-Mayer / Gaussian potential
- *bpm/spring* - repulsive harmonic force with damping
- *brownian* - Brownian potential for Fast Lubrication Dynamics
- *brownian/poly* - Brownian potential for Fast Lubrication Dynamics with polydispersity
- *buck* - Buckingham potential
- *buck/coul/cut* - Buckingham with cutoff Coulomb
- *buck/coul/long* - Buckingham with long-range Coulomb
- *buck/coul/long/cs* - Buckingham with long-range Coulomb and core/shell
- *buck/coul/msm* - Buckingham with long-range MSM Coulomb
- *buck/long/coul/long* - long-range Buckingham with long-range Coulomb
- *buck/mdf* - Buckingham with a taper function
- *buck6d/coul/gauss/dsf* - dispersion-damped Buckingham with damped-shift-force model
- *buck6d/coul/gauss/long* - dispersion-damped Buckingham with long-range Coulomb
- *colloid* - integrated colloidal potential
- *comb* - charge-optimized many-body (COMB) potential
- *comb3* - charge-optimized many-body (COMB3) potential
- *cosine/squared* - Cooke-Kremer-Deserno membrane model potential
- *coul/cut* - cutoff Coulomb potential
- *coul/cut/dielectric* -
- *coul/cut/global* - cutoff Coulomb potential
- *coul/cut/soft* - Coulomb potential with a soft core
- *coul/debye* - cutoff Coulomb potential with Debye screening
- *coul/diel* - Coulomb potential with dielectric permittivity
- *coul/dsf* - Coulomb with damped-shifted-force model
- *coul/exclude* - subtract Coulomb potential for excluded pairs
- *coul/long* - long-range Coulomb potential
- *coul/long/cs* - long-range Coulomb potential and core/shell
- *coul/long/dielectric* -
- *coul/long/soft* - long-range Coulomb potential with a soft core
- *coul/msm* - long-range MSM Coulomb
- *coul/slater/cut* - smeared out Coulomb
- *coul/slater/long* - long-range smeared out Coulomb
- *coul/shield* - Coulomb for boron nitride for use with *ilp/graphene/hbn* potential
- *coul/streitz* - Coulomb via Streitz/Mintmire Slater orbitals

- *coul/tt* - damped charge-dipole Coulomb for Drude dipoles
- *coul/wolf* - Coulomb via Wolf potential
- *coul/wolf/cs* - Coulomb via Wolf potential with core/shell adjustments
- *dpd* - dissipative particle dynamics (DPD)
- *dpd/coul/slater/long* - dissipative particle dynamics (DPD) with electrostatic interactions
- *dpd/ext* - generalized force field for DPD
- *dpd/ext/tstat* - pairwise DPD thermostatting with generalized force field
- *dpd/fdt* - DPD for constant temperature and pressure
- *dpd/fdt/energy* - DPD for constant energy and enthalpy
- *dpd/tstat* - pairwise DPD thermostatting
- *dsmc* - Direct Simulation Monte Carlo (DSMC)
- *e3b* - Explicit-three body (E3B) water model
- *drip* - Dihedral-angle-corrected registry-dependent interlayer potential (DRIP)
- *eam* - embedded atom method (EAM)
- *eam/alloy* - alloy EAM
- *eam/cd* - concentration-dependent EAM
- *eam/cd/old* - older two-site model for concentration-dependent EAM
- *eam/fs* - Finnis-Sinclair EAM
- *eam/he* - Finnis-Sinclair EAM modified for Helium in metals
- *edip* - three-body EDIP potential
- *edip/multi* - multi-element EDIP potential
- *edpd* - eDPD particle interactions
- *eff/cut* - electron force field with a cutoff
- *eim* - embedded ion method (EIM)
- *exp6/rx* - reactive DPD potential
- *extep* - extended Tersoff potential
- *gauss* - Gaussian potential
- *gauss/cut* - generalized Gaussian potential
- *gayberne* - Gay-Berne ellipsoidal potential
- *granular* - Generalized granular potential
- *gran/hertz/history* - granular potential with Hertzian interactions
- *gran/hooke* - granular potential with history effects
- *gran/hooke/history* - granular potential without history effects
- *gw* - Gao-Weber potential
- *gw/zbl* - Gao-Weber potential with a repulsive ZBL core
- *harmonic/cut* - repulsive-only harmonic potential

- *hbond/dreiding/lj* - DREIDING hydrogen bonding LJ potential
- *hbond/dreiding/morse* - DREIDING hydrogen bonding Morse potential
- *hdnnp* - High-dimensional neural network potential
- *hippo* -
- *ilp/graphene/hbn* - registry-dependent interlayer potential (ILP)
- *ilp/tmd* - interlayer potential (ILP) potential for transition metal dichalcogenides (TMD)
- *kim* - interface to potentials provided by KIM project
- *kolmogorov/crespi/full* - Kolmogorov-Crespi (KC) potential with no simplifications
- *kolmogorov/crespi/z* - Kolmogorov-Crespi (KC) potential with normals along z-axis
- *lcbop* - long-range bond-order potential (LCBOP)
- *lebedeva/z* - Lebedeva interlayer potential for graphene with normals along z-axis
- *lennard/mdf* - LJ potential in A/B form with a taper function
- *lepton* - pair potential from evaluating a string
- *lepton/coul* - pair potential from evaluating a string with support for charges
- *lepton/sphere* - pair potential from evaluating a string with support for radii
- *line/lj* - LJ potential between line segments
- *list* - potential between pairs of atoms explicitly listed in an input file
- *lj/charmm/coul/charmm* - CHARMM potential with cutoff Coulomb
- *lj/charmm/coul/charmm/implicit* - CHARMM for implicit solvent
- *lj/charmm/coul/long* - CHARMM with long-range Coulomb
- *lj/charmm/coul/long/soft* - CHARMM with long-range Coulomb and a soft core
- *lj/charmm/coul/msm* - CHARMM with long-range MSM Coulomb
- *lj/charmmfsw/coul/charmmfsh* - CHARMM with force switching and shifting
- *lj/charmmfsw/coul/long* - CHARMM with force switching and long-rnage Coulomb
- *lj/class2* - COMPASS (class 2) force field without Coulomb
- *lj/class2/coul/cut* - COMPASS with cutoff Coulomb
- *lj/class2/coul/cut/soft* - COMPASS with cutoff Coulomb with a soft core
- *lj/class2/coul/long* - COMPASS with long-range Coulomb
- *lj/class2/coul/long/cs* - COMPASS with long-range Coulomb with core/shell adjustments
- *lj/class2/coul/long/soft* - COMPASS with long-range Coulomb with a soft core
- *lj/class2/soft* - COMPASS (class 2) force field with no Coulomb with a soft core
- *lj/cubic* - LJ with cubic after inflection point
- *lj/cut* - cutoff Lennard-Jones potential without Coulomb
- *lj/cut/coul/cut* - LJ with cutoff Coulomb
- *lj/cut/coul/cut/dielectric* -
- *lj/cut/coul/cut/soft* - LJ with cutoff Coulomb with a soft core

- *lj/cut/coul/debye* - LJ with Debye screening added to Coulomb
- *lj/cut/coul/debye/dielectric* -
- *lj/cut/coul/dsf* - LJ with Coulomb via damped shifted forces
- *lj/cut/coul/long* - LJ with long-range Coulomb
- *lj/cut/coul/long/cs* - LJ with long-range Coulomb with core/shell adjustments
- *lj/cut/coul/long/dielectric* -
- *lj/cut/coul/long/soft* - LJ with long-range Coulomb with a soft core
- *lj/cut/coul/msm* - LJ with long-range MSM Coulomb
- *lj/cut/coul/msm/dielectric* -
- *lj/cut/coul/wolf* - LJ with Coulomb via Wolf potential
- *lj/cut/dipole/cut* - point dipoles with cutoff
- *lj/cut/dipole/long* - point dipoles with long-range Ewald
- *lj/cut/soft* - LJ with a soft core
- *lj/cut/sphere* - LJ where per-atom radius is used as LJ sigma
- *lj/cut/thole/long* - LJ with Coulomb with thole damping
- *lj/cut/tip4p/cut* - LJ with cutoff Coulomb for TIP4P water
- *lj/cut/tip4p/long* - LJ with long-range Coulomb for TIP4P water
- *lj/cut/tip4p/long/soft* - LJ with cutoff Coulomb for TIP4P water with a soft core
- *lj/expand* - Lennard-Jones for variable size particles
- *lj/expand/coul/long* - Lennard-Jones for variable size particles with long-range Coulomb
- *lj/expand/sphere* - Variable size LJ where per-atom radius is used as delta (size)
- *lj/gromacs* - GROMACS-style Lennard-Jones potential
- *lj/gromacs/coul/gromacs* - GROMACS-style LJ and Coulomb potential
- *lj/long/coul/long* - long-range LJ and long-range Coulomb
- *lj/long/coul/long/dielectric* -
- *lj/long/dipole/long* - long-range LJ and long-range point dipoles
- *lj/long/tip4p/long* - long-range LJ and long-range Coulomb for TIP4P water
- *lj/mdf* - LJ potential with a taper function
- *lj/relres* - LJ using multiscale Relative Resolution (RelRes) methodology *(Chaimovich)*.
- *lj/spica* - LJ for SPICA coarse-graining
- *lj/spica/coul/long* - LJ for SPICA coarse-graining with long-range Coulomb
- *lj/spica/coul/msm* - LJ for SPICA coarse-graining with long-range Coulomb via MSM
- *lj/sf/dipole/sf* - LJ with dipole interaction with shifted forces
- *lj/smooth* - smoothed Lennard-Jones potential
- *lj/smooth/linear* - linear smoothed LJ potential
- *lj/switch3/coulgauss/long* - smoothed LJ vdW potential with Gaussian electrostatics

- *lj96/cut* - Lennard-Jones 9/6 potential
- *local/density* - Generalized basic local density potential
- *lubricate* - Hydrodynamic lubrication forces
- *lubricate/poly* - Hydrodynamic lubrication forces with polydispersity
- *lubricateU* - Hydrodynamic lubrication forces for Fast Lubrication Dynamics
- *lubricateU/poly* - Hydrodynamic lubrication forces for Fast Lubrication with polydispersity
- *mdpd* - mDPD particle interactions
- *mdpd/rhosum* - mDPD particle interactions for mass density
- *meam* - Modified embedded atom method (MEAM)
- *meam/ms* - Multi-state modified embedded atom method (MS-MEAM)
- *meam/spline* - Splined version of MEAM
- *meam/sw/spline* - Splined version of MEAM with a Stillinger-Weber term
- *mesocnt* - Mesoscopic vdW potential for (carbon) nanotubes
- *mesocnt/viscous* - Mesoscopic vdW potential for (carbon) nanotubes with friction
- *mgpt* - Simplified model generalized pseudopotential theory (MGPT) potential
- *mie/cut* - Mie potential
- *mliap* - Multiple styles of machine-learning potential
- *mm3/switch3/coulgauss/long* - Smoothed MM3 vdW potential with Gaussian electrostatics
- *momb* - Many-Body Metal-Organic (MOMB) force field
- *morse* - Morse potential
- *morse/smooth/linear* - Linear smoothed Morse potential
- *morse/soft* - Morse potential with a soft core
- *multi/lucy* - DPD potential with density-dependent force
- *multi/lucy/rx* - reactive DPD potential with density-dependent force
- *nb3b/harmonic* - Non-bonded 3-body harmonic potential
- *nb3b/screened* - Non-bonded 3-body screened harmonic potential
- *nm/cut* - N-M potential
- *nm/cut/coul/cut* - N-M potential with cutoff Coulomb
- *nm/cut/coul/long* - N-M potential with long-range Coulomb
- *nm/cut/split* - Split 12-6 Lennard-Jones and N-M potential
- *oxdna/coaxstk* -
- *oxdna/excv* -
- *oxdna/hbond* -
- *oxdna/stk* -
- *oxdna/xstk* -
- *oxdna2/coaxstk* -

- *oxdna2/dh* -
- *oxdna2/excv* -
- *oxdna2/hbond* -
- *oxdna2/stk* -
- *oxdna2/xstk* -
- *oxrna2/coaxstk* -
- *oxrna2/dh* -
- *oxrna2/excv* -
- *oxrna2/hbond* -
- *oxrna2/stk* -
- *oxrna2/xstk* -
- *pace* - Atomic Cluster Expansion (ACE) machine-learning potential
- *pace/extrapolation* - Atomic Cluster Expansion (ACE) machine-learning potential with extrapolation grades
- *pedone* - Pedone (PMMCS) potential (non-Coulomb part)
- *pod* - Proper orthogonal decomposition (POD) machine-learning potential
- *peri/eps* - Peridynamic EPS potential
- *peri/lps* - Peridynamic LPS potential
- *peri/pmb* - Peridynamic PMB potential
- *peri/ves* - Peridynamic VES potential
- *polymorphic* - Polymorphic 3-body potential
- *python* -
- *quip* -
- *rann* -
- *reaxff* - ReaxFF potential
- *rebo* - Second generation REBO potential of Brenner
- *rebomos* - REBOMoS potential for MoS2
- *rheo* - fluid interactions in RHEO package
- *rheo/solid* - solid interactions in RHEO package
- *resquared* - Everaers RE-Squared ellipsoidal potential
- *saip/metal* - Interlayer potential for hetero-junctions formed with hexagonal 2D materials and metal surfaces
- *sdpd/taitwater/isothermal* - Smoothed dissipative particle dynamics for water at isothermal conditions
- *smatb* - Second Moment Approximation to the Tight Binding
- *smatb/single* - Second Moment Approximation to the Tight Binding for single-element systems
- *smd/hertz* -
- *smd/tlsph* -
- *smd/tri_surface* -

- *smd/ulsph* -
- *smtbq* -
- *snap* - SNAP machine-learning potential
- *soft* - Soft (cosine) potential
- *sph/heatconduction* -
- *sph/idealgas* -
- *sph/lj* -
- *sph/rhosum* -
- *sph/taitwater* -
- *sph/taitwater/morris* -
- *spin/dipole/cut* -
- *spin/dipole/long* -
- *spin/dmi* -
- *spin/exchange* -
- *spin/exchange/biquadratic* -
- *spin/magelec* -
- *spin/neel* -
- *srp* -
- *srp/react* -
- *sw* - Stillinger-Weber 3-body potential
- *sw/angle/table* - Stillinger-Weber potential with tabulated angular term
- *sw/mod* - modified Stillinger-Weber 3-body potential
- *table* - tabulated pair potential
- *table/rx* -
- *tdpd* - tDPD particle interactions
- *tersoff* - Tersoff 3-body potential
- *tersoff/mod* - modified Tersoff 3-body potential
- *tersoff/mod/c* -
- *tersoff/table* -
- *tersoff/zbl* - Tersoff/ZBL 3-body potential
- *thole* - Coulomb interactions with thole damping
- *threebody/table* - generic tabulated three-body potential
- *tip4p/cut* - Coulomb for TIP4P water w/out LJ
- *tip4p/long* - long-range Coulomb for TIP4P water w/out LJ
- *tip4p/long/soft* -
- *tracker* - monitor information about pairwise interactions

- *tri/lj* - LJ potential between triangles
- *ufm* -
- *uf3* - UF3 machine-learning potential
- *vashishta* - Vashishta 2-body and 3-body potential
- *vashishta/table* -
- *wf/cut* - Wang-Frenkel Potential for short-ranged interactions
- *ylz* - Yuan-Li-Zhang Potential for anisotropic interactions
- *yukawa* - Yukawa potential
- *yukawa/colloid* - screened Yukawa potential for finite-size particles
- *zbl* - Ziegler-Biersack-Littmark potential

1.76.4 Restrictions

This command must be used before any coefficients are set by the *pair_coeff*, *read_data*, or *read_restart* commands.

Some pair styles are part of specific packages. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info. The doc pages for individual pair potentials tell if it is part of a package.

1.76.5 Related commands

pair_coeff, *read_data*, *pair_modify*, *kspace_style*, *dielectric*, *pair_write*

1.76.6 Default

```
pair_style none
```

1.77 pair_write command

1.77.1 Syntax

```
pair_write itype jtype N style inner outer file keyword Qi Qj
```

- itype,jtype = 2 atom types (numeric or type label)
- N = # of values
- style = *r* or *rsq* or *bitmap*
- inner,outer = inner and outer cutoff (distance units)
- file = name of file to write values to
- keyword = section name in file for this set of tabulated values
- Qi,Qj = 2 atom charges (charge units) (optional)

1.77.2 Examples

```
pair_write 1 3 500 r 1.0 10.0 table.txt LJ
pair_write 1 1 1000 rsq 2.0 8.0 table.txt Yukawa_1_1 -0.5 0.5

labelmap atom 1 C 2 H
pair_write C H 500 r 1.0 10.0 table.txt LJ
```

1.77.3 Description

Write energy and force values to a file as a function of distance for the currently defined pair potential. This is useful for plotting the potential function or otherwise debugging its values. If the file already exists, the table of values is appended to the end of the file to allow multiple tables of energy and force to be included in one file. In case a new file is created, the first line will be a comment containing a "DATE:" and "UNITS:" tag with the current date and the current *units* setting as argument. For subsequent invocations of the pair_write command, the current units setting is compared against the entry in the file, if present, and pair_write will refuse to add a table if the units are not the same.

The energy and force values are computed at distances from inner to outer for 2 interacting atoms of type *itype* and *jtype*, using the appropriate *pair_coeff* coefficients. If the style is *r*, then N distances are used, evenly spaced in r; if the style is *rsq*, N distances are used, evenly spaced in r^2.

For example, for N = 7, style = *r*, inner = 1.0, and outer = 4.0, values are computed at r = 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0.

If the style is *bitmap*, then 2^N values are written to the file in a format and order consistent with how they are read in by the *pair_coeff* command for pair style *table*. For reasonable accuracy in a bitmapped table, choose N >= 12, an *inner* value that is smaller than the distance of closest approach of 2 atoms, and an *outer* value <= cutoff of the potential.

If the pair potential is computed between charged atoms, the charges of the pair of interacting atoms can optionally be specified. If not specified, values of Qi = Qj = 1.0 are used.

The file is written in the format used as input for the *pair_style table* option with *keyword* as the section name. Each line written to the file lists an index number (1-N), a distance (in distance units), an energy (in energy units), and a force (in force units).

1.77.4 Restrictions

All force field coefficients for pair and other kinds of interactions must be set before this command can be invoked.

Due to how the pairwise force is computed, an inner value > 0.0 must be specified even if the potential has a finite value at r = 0.0.

The *pair_write* command can only be used for pairwise additive interactions for which a *Pair::single()* function can be and has been implemented. This excludes for example manybody potentials or TIP4P coulomb styles.

1.77.5 Related commands

pair_style table, *pair_style*, *pair_coeff*

1.77.6 Default

none

1.78 partition command

1.78.1 Syntax

```
partition style N command ...
```

- style = *yes* or *no*
- N = partition number (see asterisk form below)
- command = any LAMMPS command

1.78.2 Examples

```
partition yes 1 processors 4 10 6
partition no 5 print "Active partition"
partition yes *5 fix all nve
partition yes 6* fix all nvt temp 1.0 1.0 0.1
```

1.78.3 Description

This command invokes the specified command on a subset of the partitions of processors you have defined via the *-partition command-line switch*.

Normally, every input script command in your script is invoked by every partition. This behavior can be modified by defining world- or universe-style *variables* that have different values for each partition. This mechanism can be used to cause your script to jump to different input script files on different partitions, if such a variable is used in a *jump* command.

The "partition" command is another mechanism for having as input script operate differently on different partitions. It is basically a prefix on any LAMMPS command. The command will only be invoked on the partition(s) specified by the *style* and *N* arguments.

If the *style* is *yes*, the command will be invoked on any partition which matches the *N* argument. If the *style* is *no* the command will be invoked on all the partitions which do not match the Np argument.

Partitions are numbered from 1 to Np, where Np is the number of partitions specified by the *-partition command-line switch*.

N can be specified in one of two ways. An explicit numeric value can be used, as in the first example above. Or a wild-card asterisk can be used to span a range of partition numbers. This takes the form "*" or "*n" or "n*" or "m*n". An asterisk with no numeric values means all partitions from 1 to Np. A leading asterisk means all partitions from 1 to n (inclusive). A trailing asterisk means all partitions from n to Np (inclusive). A middle asterisk means all partitions from m to n (inclusive).

This command can be useful for the "run_style verlet/split" command which imposed requirements on how the *processors* command lays out a 3d grid of processors in each of 2 partitions.

1.78.4 Restrictions

none

1.78.5 Related commands

run_style verlet/split

1.78.6 Default

none

1.79 plugin command

1.79.1 Syntax

```
plugin command args
```

- command = *load* or *unload* or *list* or *clear*
- args = list of arguments for a particular plugin command

  load file = load plugin(s) from shared object in file
  unload style name = unload plugin name of style style
    style = pair or bond or angle or dihedral or improper or kspace or compute or fix or region or command
  list = print a list of currently loaded plugins
  clear = unload all currently loaded plugins

1.79.2 Examples

```
plugin load morse2plugin.so
plugin unload pair morse2/omp
plugin unload command hello
plugin list
plugin clear
```

1.79.3 Description

The plugin command allows to load (and unload) additional styles and commands into a LAMMPS binary from so-called dynamic shared object (DSO) files. This enables to add new functionality to an existing LAMMPS binary without having to recompile and link the entire executable.

The *load* command will load and initialize all plugins contained in the plugin DSO with the given filename. A message with information the plugin style and name and more will be printed. Individual DSO files may contain multiple plugins. More details about how to write and compile the plugin DSO is given in programmer's guide part of the manual under *Writing plugins*.

The *unload* command will remove the given style or the given name from the list of available styles. If the plugin style is currently in use, that style instance will be deleted.

The *list* command will print a list of the loaded plugins and their styles and names.

The *clear* command will unload all currently loaded plugins.

Automatic loading of plugins

Added in version 4May2022.

When the environment variable LAMMPS_PLUGIN_PATH is set, then LAMMPS will search the directory (or directories) listed in this path for files with names that end in plugin.so (e.g. helloplugin.so) and will try to load the contained plugins automatically at start-up.

1.79.4 Restrictions

The *plugin* command is part of the PLUGIN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

If plugins access functions or classes from a package, LAMMPS must have been compiled with that package included.

Plugins are dependent on the LAMMPS binary interface (ABI) and particularly the MPI library used. So they are not guaranteed to work when the plugin was compiled with a different MPI library or different compilation settings or a different LAMMPS version. There are no checks, so if there is a mismatch the plugin object will either not load or data corruption and crashes may happen.

1.79.5 Related commands

none

1.79.6 Default

none

1.80 prd command

1.80.1 Syntax

```
prd N t_event n_dephase t_dephase t_correlate compute-ID seed keyword value ...
```

- N = # of timesteps to run (not including dephasing/quenching)
- t_event = timestep interval between event checks
- n_dephase = number of velocity randomizations to perform in each dephase run
- t_dephase = number of timesteps to run dynamics after each velocity randomization during dephase
- t_correlate = number of timesteps within which 2 consecutive events are considered to be correlated
- compute-ID = ID of the compute used for event detection
- random_seed = random # seed (positive integer)

- zero or more keyword/value pairs may be appended
- keyword = *min* or *temp* or *vel* or *time*

  min values = etol ftol maxiter maxeval
    etol = stopping tolerance for energy, used in quenching
    ftol = stopping tolerance for force, used in quenching
    maxiter = max iterations of minimize, used in quenching
    maxeval = max number of force/energy evaluations, used in quenching
  temp value = Tdephase
    Tdephase = target temperature for velocity randomization, used in dephasing
  vel values = loop dist
    loop = all or local or geom, used in dephasing
    dist = uniform or gaussian, used in dephasing
  time value = steps or clock
    steps = simulation runs for N timesteps on each replica (default)
    clock = simulation runs for N timesteps across all replicas

1.80.2 Examples

```
prd 5000 100 10 10 100 1 54982
prd 5000 100 10 10 100 1 54982 min 0.1 0.1 100 200
```

1.80.3 Description

Run a parallel replica dynamics (PRD) simulation using multiple replicas of a system. One or more replicas can be used. The total number of steps *N* to run can be interpreted in one of two ways; see discussion of the *time* keyword below.

PRD is described in *(Voter1998)* by Art Voter. Similar to global or local hyperdynamics (HD), PRD is a method for performing accelerated dynamics that is suitable for infrequent-event systems that obey first-order kinetics. A good overview of accelerated dynamics methods (AMD) for such systems in given in this review paper *(Voter2002)* from Art's group. To quote from the paper: "The dynamical evolution is characterized by vibrational excursions within a potential basin, punctuated by occasional transitions between basins. The transition probability is characterized by p(t) = k*exp(-kt) where k is the rate constant."

Both PRD and HD produce a time-accurate trajectory that effectively extends the timescale over which a system can be simulated, but they do it differently. PRD creates Nr replicas of the system and runs dynamics on each independently with a normal unbiased potential until an event occurs in one of the replicas. The time between events is reduced by a factor of Nr replicas. HD uses a single replica of the system and accelerates time by biasing the interaction potential in a manner such that each timestep is effectively longer. For both methods, per CPU second, more physical time elapses and more events occur. See the *hyper* page for more info about HD.

In PRD, each replica runs on a partition of one or more processors. Processor partitions are defined at run-time using the *-partition command-line switch*. Note that if you have MPI installed, you can run a multi-replica simulation with more replicas (partitions) than you have physical processors, e.g you can run a 10-replica simulation on one or two processors. However for PRD, this makes little sense, since running a replica on virtual instead of physical processors,offers no effective parallel speed-up in searching for infrequent events. See the *Howto replica* doc page for further discussion.

When a PRD simulation is performed, it is assumed that each replica is running the same model, though LAMMPS does not check for this. I.e. the simulation domain, the number of atoms, the interaction potentials, etc should be the same for every replica.

A PRD run has several stages, which are repeated each time an "event" occurs in one of the replicas, as explained below. The logic for a PRD run is as follows:

```
while (time remains):
  dephase for n_dephase*t_dephase steps
  until (event occurs on some replica):
    run dynamics for t_event steps
    quench
    check for uncorrelated event on any replica
  until (no correlated event occurs):
    run dynamics for t_correlate steps
    quench
    check for correlated event on this replica
  event replica shares state with all replicas
```

Before this loop begins, the state of the system on replica 0 is shared with all replicas, so that all replicas begin from the same initial state. The first potential energy basin is identified by quenching (an energy minimization, see below) the initial state and storing the resulting coordinates for reference.

In the first stage, dephasing is performed by each replica independently to eliminate correlations between replicas. This is done by choosing a random set of velocities, based on the *random_seed* that is specified, and running *t_dephase* timesteps of dynamics. This is repeated *n_dephase* times. At each of the *n_dephase* stages, if an event occurs during the *t_dephase* steps of dynamics for a particular replica, the replica repeats the stage until no event occurs.

If the *temp* keyword is not specified, the target temperature for velocity randomization for each replica is the current temperature of that replica. Otherwise, it is the specified *Tdephase* temperature. The style of velocity randomization is controlled using the keyword *vel* with arguments that have the same meaning as their counterparts in the *velocity* command.

In the second stage, each replica runs dynamics continuously, stopping every *t_event* steps to check if a transition event has occurred. This check is performed by quenching the system and comparing the resulting atom coordinates to the coordinates from the previous basin. The first time through the PRD loop, the "previous basin" is the set of quenched coordinates from the initial state of the system.

A quench is an energy minimization and is performed by whichever algorithm has been defined by the *min_style* command. Minimization parameters may be set via the *min_modify* command and by the *min* keyword of the PRD command. The latter are the settings that would be used with the *minimize* command. Note that typically, you do not need to perform a highly-converged minimization to detect a transition event, though you may need to in order to prevent a set of atoms in the system from relaxing to a saddle point.

The event check is performed by a compute with the specified *compute-ID*. Currently there is only one compute that works with the PRD command, which is the *compute event/displace* command. Other event-checking computes may be added. *Compute event/displace* checks whether any atom in the compute group has moved further than a specified threshold distance. If so, an "event" has occurred.

In the third stage, the replica on which the event occurred (event replica) continues to run dynamics to search for correlated events. This is done by running dynamics for *t_correlate* steps, quenching every *t_event* steps, and checking if another event has occurred.

The first time no correlated event occurs, the final state of the event replica is shared with all replicas, the new basin reference coordinates are updated with the quenched state, and the outer loop begins again. While the replica event is searching for correlated events, all the other replicas also run dynamics and event checking with the same schedule, but the final states are always overwritten by the state of the event replica.

The outer loop of the pseudocode above continues until *N* steps of dynamics have been performed. Note that *N* only includes the dynamics of stages 2 and 3, not the steps taken during dephasing or the minimization iterations of quenching. The specified *N* is interpreted in one of two ways, depending on the *time* keyword. If the *time* value is *steps*, which is the default, then each replica runs for *N* timesteps. If the *time* value is *clock*, then the simulation runs until *N* aggregate timesteps across all replicas have elapsed. This aggregate time is the "clock" time defined below, which typically advances nearly M times faster than the timestepping on a single replica, where M is the number of replicas.

Four kinds of output can be generated during a PRD run: event statistics, thermodynamic output by each replica, dump files, and restart files.

When running with multiple partitions (each of which is a replica in this case), the print-out to the screen and master log.lammps file is limited to event statistics. Note that if a PRD run is performed on only a single replica then the event statistics will be intermixed with the usual thermodynamic output discussed below.

The quantities printed each time an event occurs are the timestep, CPU time, clock, event number, a correlation flag, the number of coincident events, and the replica number of the chosen event.

The timestep is the usual LAMMPS timestep, except that time does not advance during dephasing or quenches, but only during dynamics. Note that are two kinds of dynamics in the PRD loop listed above that contribute to this timestepping. The first is when all replicas are performing independent dynamics, waiting for an event to occur. The second is when correlated events are being searched for, but only one replica is running dynamics.

The CPU time is the total elapsed time on each processor, since the start of the PRD run.

The clock is the same as the timestep except that it advances by M steps per timestep during the first kind of dynamics when the M replicas are running independently. The clock advances by only 1 step per timestep during the second kind of dynamics, when only a single replica is checking for a correlated event. Thus "clock" time represents the aggregate time (in steps) that has effectively elapsed during a PRD simulation on M replicas. If most of the PRD run is spent in the second stage of the loop above, searching for infrequent events, then the clock will advance nearly M times faster than it would if a single replica was running. Note the clock time between successive events should be drawn from p(t).

The event number is a counter that increments with each event, whether it is uncorrelated or correlated.

The correlation flag will be 0 when an uncorrelated event occurs during the second stage of the loop listed above, i.e. when all replicas are running independently. The correlation flag will be 1 when a correlated event occurs during the third stage of the loop listed above, i.e. when only one replica is running dynamics.

When more than one replica detects an event at the end of the same event check (every *t_event* steps) during the second stage, then one of them is chosen at random. The number of coincident events is the number of replicas that detected an event. Normally, this value should be 1. If it is often greater than 1, then either the number of replicas is too large, or *t_event* is too large.

The replica number is the ID of the replica (from 0 to M-1) in which the event occurred.

When running on multiple partitions, LAMMPS produces additional log files for each partition, e.g. log.lammps.0, log.lammps.1, etc. For the PRD command, these contain the thermodynamic output for each replica. You will see short runs and minimizations corresponding to the dynamics and quench operations of the loop listed above. The timestep will be reset appropriately depending on whether the operation advances time or not.

After the PRD command completes, timing statistics for the PRD run are printed in each replica's log file, giving a breakdown of how much CPU time was spent in each stage (dephasing, dynamics, quenching, etc).

Any *dump files* defined in the input script, will be written to during a PRD run at timesteps corresponding to both uncorrelated and correlated events. This means the requested dump frequency in the *dump* command is ignored. There will be one dump file (per dump command) created for all partitions.

The atom coordinates of the dump snapshot are those of the minimum energy configuration resulting from quenching following a transition event. The timesteps written into the dump files correspond to the timestep at which the event occurred and NOT the clock. A dump snapshot corresponding to the initial minimum state used for event detection is written to the dump file at the beginning of each PRD run.

If the *restart* command is used, a single restart file for all the partitions is generated, which allows a PRD run to be continued by a new input script in the usual manner.

The restart file is generated at the end of the loop listed above. If no correlated events are found, this means it contains a snapshot of the system at time T + *t_correlate*, where T is the time at which the uncorrelated event occurred. If correlated events were found, then it contains a snapshot of the system at time T + *t_correlate*, where T is the time of the last correlated event.

The restart frequency specified in the *restart* command is interpreted differently when performing a PRD run. It does not mean the timestep interval between restart files. Instead it means an event interval for uncorrelated events. Thus a frequency of 1 means write a restart file every time an uncorrelated event occurs. A frequency of 10 means write a restart file every 10th uncorrelated event.

When an input script reads a restart file from a previous PRD run, the new script can be run on a different number of replicas or processors. However, it is assumed that *t_correlate* in the new PRD command is the same as it was previously. If not, the calculation of the "clock" value for the first event in the new run will be slightly off.

1.80.4 Restrictions

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

The *N* and *t_correlate* settings must be integer multiples of *t_event*.

Runs restarted from restart file written during a PRD run will not produce identical results due to changes in the random numbers used for dephasing.

This command cannot be used when any fixes are defined that keep track of elapsed time to perform time-dependent operations. Examples include the "ave" fixes such as *fix ave/chunk*. Also *fix dt/reset* and *fix deposit*.

1.80.5 Related commands

compute event/displace, *min_modify*, *min_style*, *run_style*, *minimize*, *velocity*, *temper*, *neb*, *tad*, *hyper*

1.80.6 Default

The option defaults are min = 0.1 0.1 40 50, no temp setting, vel = geom gaussian, and time = steps.

(Voter1998) Voter, Phys Rev B, 57, 13985 (1998).

(Voter2002) Voter, Montalenti, Germann, Annual Review of Materials Research 32, 321 (2002).

1.81 print command

1.81.1 Syntax

```
print string keyword value
```

- string = text string to print, which may contain variables
- zero or more keyword/value pairs may be appended
- keyword = *file* or *append* or *screen* or *universe*

file value = filename
append value = filename
screen value = yes or no
universe value = yes or no

1.81.2 Examples

```
print "Done with equilibration" file info.dat
print Vol=$v append info.dat screen no
print "The system volume is now $v"
print 'The system volume is now $v'
print "NEB calculation 1 complete" screen no universe yes
print """
System volume = $v
System temperature = $t
"""
```

1.81.3 Description

Print a text string to the screen and logfile. The text string must be a single argument, so if it is one line but more than one word, it should be enclosed in single or double quotes. To generate multiple lines of output, the string can be enclosed in triple quotes, as in the last example above. If the text string contains variables, they will be evaluated and their current values printed.

Added in version 15Jun2023: support for vector style variables

See the *variable* command for a description of *equal* and *vector* style variables which are typically the most useful ones to use with the print command. Equal- and vector-style variables can calculate formulas involving mathematical operations, atom properties, group properties, thermodynamic properties, global values calculated by a *compute* or *fix*, or references to other *variables*. Vector-style variables are printed in a bracketed, comma-separated format, e.g. [1,2,3,4] or [12.5,2,4.6,10.1].

Note

As discussed on the *Commands parse* doc page, the text string can use "immediate" variables, specified as $(formula) with parenthesis, where the numeric formula has the same syntax as equal-style variables described on the *variable* doc page. This is a convenient way to evaluate a formula immediately without using the variable command to define a named variable and then use that variable in the text string. The formula can include a trailing colon and format string which determines the precision with which the numeric value is output. This is also explained on the *Commands parse* doc page.

If you want the print command to be executed multiple times (with changing variable values), there are 3 options. First, consider using the *fix print* command, which will print a string periodically during a simulation. Second, the print command can be used as an argument to the *every* option of the *run* command. Third, the print command could appear in a section of the input script that is looped over (see the *jump* and *next* commands).

If the *file* or *append* keyword is used, a filename is specified to which the output will be written. If *file* is used, then the filename is overwritten if it already exists. If *append* is used, then the filename is appended to if it already exists, or created if it does not exist.

If the *screen* keyword is used, output to the screen and logfile can be turned on or off as desired.

If the *universe* keyword is used, output to the global screen and logfile can be turned on or off as desired. In multi-partition calculations, the *screen* option and the corresponding output only apply to the screen and logfile of the individual partition.

1.81.4 Restrictions

none

1.81.5 Related commands

fix print, *variable*

1.81.6 Default

The option defaults are no file output, screen = yes, and universe = no.

1.82 processors command

1.82.1 Syntax

```
processors Px Py Pz keyword args ...
```

- Px,Py,Pz = # of processors in each dimension of 3d grid overlaying the simulation domain
- zero or more keyword/arg pairs may be appended
- keyword = *grid* or *map* or *part* or *file*

```
grid arg = gstyle params ...
  gstyle = onelevel or twolevel or numa or custom
    onelevel params = none
    twolevel params = Nc Cx Cy Cz
      Nc = number of cores per node
      Cx,Cy,Cz = # of cores in each dimension of 3d sub-grid assigned to each node
    numa params = none
    custom params = infile
      infile = file containing grid layout
numa_nodes arg = Nn
      Nn = number of numa domains per node
map arg = cart or cart/reorder or xyz or xzy or yxz or yzx or zxy or zyx
   cart = use MPI_Cart() methods to map processors to 3d grid with reorder = 0
   cart/reorder = use MPI_Cart() methods to map processors to 3d grid with reorder = 1
   xyz,xzy,yxz,yzx,zxy,zyx = map processors to 3d grid in IJK ordering
part args = Psend Precv cstyle
  Psend = partition # (1 to Np) which will send its processor layout
  Precv = partition # (1 to Np) which will recv the processor layout
  cstyle = multiple
    multiple = Psend grid will be multiple of Precv grid in each dimension
file arg = outfile
  outfile = name of file to write 3d grid of processors to
```

1.82.2 Examples

```
processors * * 5
processors 2 4 4
processors * * 8 map xyz
processors * * * grid numa
processors * * * grid twolevel 4 * * 1
processors 4 8 16 grid custom myfile
processors * * * part 1 2 multiple
```

1.82.3 Description

Specify how processors are mapped as a regular 3d grid to the global simulation box. The mapping involves 2 steps. First if there are P processors it means choosing a factorization P = Px by Py by Pz so that there are Px processors in the x dimension, and similarly for the y and z dimensions. Second, the P processors are mapped to the regular 3d grid. The arguments to this command control each of these 2 steps.

The Px, Py, Pz parameters affect the factorization. Any of the 3 parameters can be specified with an asterisk "*", which means LAMMPS will choose the number of processors in that dimension of the grid. It will do this based on the size and shape of the global simulation box so as to minimize the surface-to-volume ratio of each processor's subdomain.

Choosing explicit values for Px or Py or Pz can be used to override the default manner in which LAMMPS will create the regular 3d grid of processors, if it is known to be sub-optimal for a particular problem. E.g. a problem where the extent of atoms will change dramatically in a particular dimension over the course of the simulation.

The product of Px, Py, Pz must equal P, the total # of processors LAMMPS is running on. For a *2d simulation*, Pz must equal 1.

Note that if you run on a prime number of processors P, then a grid such as 1 x P x 1 will be required, which may incur extra communication costs due to the high surface area of each processor's subdomain.

Also note that if multiple partitions are being used then P is the number of processors in this partition; see the *-partition command-line switch* page for details. Also note that you can prefix the processors command with the *partition* command to easily specify different Px,Py,Pz values for different partitions.

You can use the *partition* command to specify different processor grids for different partitions, e.g.

```
partition yes 1 processors 4 4 4
partition yes 2 processors 2 3 2
```

Note

This command only affects the initial regular 3d grid created when the simulation box is first specified via a *create_box* or *read_data* or *read_restart* command. Or if the simulation box is re-created via the *replicate* command. The same regular grid is initially created, regardless of which *comm_style* command is in effect.

If load-balancing is never invoked via the *balance* or *fix balance* commands, then the initial regular grid will persist for all simulations. If balancing is performed, some of the methods invoked by those commands retain the logical topology of the initial 3d grid, and the mapping of processors to the grid specified by the processors command. However the grid spacings in different dimensions may change, so that processors own subdomains of different sizes. If the *comm_style tiled* command is used, methods invoked by the balancing commands may discard the 3d grid of processors and tile the simulation domain with subdomains of different sizes and shapes which no longer have a logical 3d connectivity. If that occurs, all the information specified by the processors command is ignored.

The *grid* keyword affects the factorization of P into Px,Py,Pz and it can also affect how the P processor IDs are mapped to the 3d grid of processors.

The *onelevel* style creates a 3d grid that is compatible with the Px,Py,Pz settings, and which minimizes the surface-to-volume ratio of each processor's subdomain, as described above. The mapping of processors to the grid is determined by the *map* keyword setting.

The *twolevel* style can be used on machines with multicore nodes to minimize off-node communication. It ensures that contiguous subsections of the 3d grid are assigned to all the cores of a node. For example if *Nc* is 4, then 2x2x1 or 2x1x2 or 1x2x2 subsections of the 3d grid will correspond to the cores of each node. This affects both the factorization and mapping steps.

The *Cx*, *Cy*, *Cz* settings are similar to the *Px*, *Py*, *Pz* settings, only their product should equal *Nc*. Any of the 3 parameters can be specified with an asterisk "*", which means LAMMPS will choose the number of cores in that dimension of the node's sub-grid. As with Px,Py,Pz, it will do this based on the size and shape of the global simulation box so as to minimize the surface-to-volume ratio of each processor's subdomain.

Note

For the *twolevel* style to work correctly, it assumes the MPI ranks of processors LAMMPS is running on are ordered by core and then by node. E.g. if you are running on 2 quad-core nodes, for a total of 8 processors, then it assumes processors 0,1,2,3 are on node 1, and processors 4,5,6,7 are on node 2. This is the default rank ordering for most MPI implementations, but some MPIs provide options for this ordering, e.g. via environment variable settings.

The *numa* style operates similar to the *twolevel* keyword except that it auto-detects which cores are running on which nodes. It will also subdivide the cores into numa domains. Currently, the number of numa domains is not auto-detected and must be specified using the *numa_nodes* keyword; otherwise, the default value is used. The *numa* style uses a different algorithm than the *twolevel* keyword for doing the two-level factorization of the simulation box into a 3d processor grid to minimize off-node communication and communication across numa domains. It does its own MPI-based mapping of nodes and cores to the regular 3d grid. Thus it may produce a different layout of the processors than the *twolevel* options.

The *numa* style will give an error if the number of MPI processes is not divisible by the number of cores used per node, or any of the Px or Py or Pz values is greater than 1.

Note

Unlike the *twolevel* style, the *numa* style does not require any particular ordering of MPI ranks in order to work correctly. This is because it auto-detects which processes are running on which nodes. However, it assumes that the lowest ranks are in the first numa domain, and so forth. MPI rank orderings that do not preserve this property might result in more intra-node communication between CPUs.

The *custom* style uses the file *infile* to define both the 3d factorization and the mapping of processors to the grid.

The file should have the following format. Any number of initial blank or comment lines (starting with a "#" character) can be present. The first non-blank, non-comment line should have 3 values:

```
Px Py Py
```

These must be compatible with the total number of processors and the Px, Py, Pz settings of the processors command.

This line should be immediately followed by P = Px*Py*Pz lines of the form:

```
ID I J K
```

where ID is a processor ID (from 0 to P-1) and I,J,K are the processors location in the 3d grid. I must be a number from 1 to Px (inclusive) and similarly for J and K. The P lines can be listed in any order, but no processor ID should appear more than once.

The *numa_nodes* keyword is used to specify the number of numa domains per node. It is currently only used by the *numa* style for two-level factorization to reduce the amount of MPI communications between CPUs. A good setting for this will typically be equal to the number of CPU sockets per node.

The *map* keyword affects how the P processor IDs (from 0 to P-1) are mapped to the 3d grid of processors. It is only used by the *onelevel* and *twolevel* grid settings.

The *cart* style uses the family of MPI Cartesian functions to perform the mapping, namely MPI_Cart_create(), MPI_Cart_get(), MPI_Cart_shift(), and MPI_Cart_rank(). It invokes the MPI_Cart_create() function with its reorder flag = 0, so that MPI is not free to reorder the processors.

The *cart/reorder* style does the same thing as the *cart* style except it sets the reorder flag to 1, so that MPI can reorder processors if it desires.

The *xyz*, *xzy*, *yxz*, *yzx*, *zxy*, and *zyx* styles are all similar. If the style is IJK, then it maps the P processors to the grid so that the processor ID in the I direction varies fastest, the processor ID in the J direction varies next fastest, and the processor ID in the K direction varies slowest. For example, if you select style *xyz* and you have a 2x2x2 grid of 8 processors, the assignments of the 8 octants of the simulation domain will be:

```
proc 0 = lo x, lo y, lo z octant
proc 1 = hi x, lo y, lo z octant
proc 2 = lo x, hi y, lo z octant
proc 3 = hi x, hi y, lo z octant
proc 4 = lo x, lo y, hi z octant
proc 5 = hi x, lo y, hi z octant
proc 6 = lo x, hi y, hi z octant
proc 7 = hi x, hi y, hi z octant
```

Note that, in principle, an MPI implementation on a particular machine should be aware of both the machine's network topology and the specific subset of processors and nodes that were assigned to your simulation. Thus its MPI_Cart calls can optimize the assignment of MPI processes to the 3d grid to minimize communication costs. In practice, however, few if any MPI implementations actually do this. So it is likely that the *cart* and *cart/reorder* styles simply give the same result as one of the IJK styles.

Also note, that for the *twolevel* grid style, the *map* setting is used to first map the nodes to the 3d grid, then again to the cores within each node. For the latter step, the *cart* and *cart/reorder* styles are not supported, so an *xyz* style is used in their place.

The *part* keyword affects the factorization of P into Px,Py,Pz.

It can be useful when running in multi-partition mode, e.g. with the *run_style verlet/split* command. It specifies a dependency between a sending partition *Psend* and a receiving partition *Precv* which is enforced when each is setting up their own mapping of their processors to the simulation box. Each of *Psend* and *Precv* must be integers from 1 to Np, where Np is the number of partitions you have defined via the *-partition command-line switch*.

A "dependency" means that the sending partition will create its regular 3d grid as Px by Py by Pz and after it has done this, it will send the Px,Py,Pz values to the receiving partition. The receiving partition will wait to receive these values before creating its own regular 3d grid and will use the sender's Px,Py,Pz values as a constraint. The nature of the constraint is determined by the *cstyle* argument.

For a *cstyle* of *multiple*, each dimension of the sender's processor grid is required to be an integer multiple of the corresponding dimension in the receiver's processor grid. This is a requirement of the *run_style verlet/split* command.

For example, assume the sending partition creates a 4x6x10 grid = 240 processor grid. If the receiving partition is running on 80 processors, it could create a 4x2x10 grid, but it will not create a 2x4x10 grid, since in the y-dimension, 6 is not an integer multiple of 4.

Note

If you use the *partition* command to invoke different "processors" commands on different partitions, and you also use the *part* keyword, then you must ensure that both the sending and receiving partitions invoke the "processors" command that connects the 2 partitions via the *part* keyword. LAMMPS cannot easily check for this, but your simulation will likely hang in its setup phase if this error has been made.

The *file* keyword writes the mapping of the factorization of P processors and their mapping to the 3d grid to the specified file *outfile*. This is useful to check that you assigned physical processors in the manner you desired, which can be tricky to figure out, especially when running on multiple partitions or on, a multicore machine or when the processor ranks were reordered by use of the *-reorder command-line switch* or due to use of MPI-specific launch options such as a config file.

If you have multiple partitions you should ensure that each one writes to a different file, e.g. using a *world-style variable* for the filename. The file has a self-explanatory header, followed by one-line per processor in this format:

world-ID universe-ID original-ID: I J K: name

The IDs are the processor's rank in this simulation (the world), the universe (of multiple simulations), and the original MPI communicator used to instantiate LAMMPS, respectively. The world and universe IDs will only be different if you are running on more than one partition; see the *-partition command-line switch*. The universe and original IDs will only be different if you used the *-reorder command-line switch* to reorder the processors differently than their rank in the original communicator LAMMPS was instantiated with.

I,J,K are the indices of the processor in the regular 3d grid, each from 1 to Nd, where Nd is the number of processors in that dimension of the grid.

The *name* is what is returned by a call to MPI_Get_processor_name() and should represent an identifier relevant to the physical processors in your machine. Note that depending on the MPI implementation, multiple cores can have the same *name*.

1.82.4 Restrictions

This command cannot be used after the simulation box is defined by a *read_data* or *create_box* command. It can be used before a restart file is read to change the 3d processor grid from what is specified in the restart file.

The *grid numa* keyword only currently works with the *map cart* option.

The *part* keyword (for the receiving partition) only works with the *grid onelevel* or *grid twolevel* options.

1.82.5 Related commands

partition, *-reorder command-line switch*

1.82.6 Default

The option defaults are Px Py Pz = * * *, grid = onelevel, map = cart, and numa_nodes = 2.

1.83 python command

1.83.1 Syntax

```
python mode keyword args ...
```

- mode = *source* or name of Python function

  if mode is *source*:

  keyword = here or name of a Python file
    here arg = inline
      inline = one or more lines of Python code which defines func
               must be a single argument, typically enclosed between triple quotes
    Python file = name of a file with Python code which will be executed immediately

- if *mode* is the name of a Python function, one or more keywords with/without arguments must be appended

  keyword = invoke or input or return or format or length or file or here or exists
    invoke arg = none = invoke the previously defined Python function
    input args = N i1 i2 ... iN
      N = # of inputs to function
      i1,...,iN = value, SELF, or LAMMPS variable name
        value = integer number, floating point number, or string
        SELF = reference to LAMMPS itself which can be accessed by Python function
        variable = v_name, where name = name of LAMMPS variable, e.g. v_abc
    return arg = varReturn
      varReturn = v_name = LAMMPS variable name which the return value of the Python function will be assigned to
    format arg = fstring with M characters
      M = N if no return value, where N = # of inputs
      M = N+1 if there is a return value
      fstring = each character (i,f,s,p) corresponds in order to an input or return value
      'i' = integer, 'f' = floating point, 's' = string, 'p' = SELF
    length arg = Nlen
      Nlen = max length of string returned from Python function
    file arg = filename
      filename = file of Python code, which defines func
    here arg = inline
      inline = one or more lines of Python code which defines func
               must be a single argument, typically enclosed between triple quotes
    exists arg = none = Python code has been loaded by previous python command

1.83.2 Examples

```
python pForce input 2 v_x 20.0 return v_f format fff file force.py
python pForce invoke

python factorial input 1 myN return v_fac format ii here """
def factorial(n):
  if n == 1: return n
  return n * factorial(n-1)
 """

python loop input 1 SELF return v_value format pf here """
def loop(lmpptr,N,cut0):
  from lammps import lammps
  lmp = lammps(ptr=lmpptr)

  # loop N times, increasing cutoff each time

  for i in range(N):
    cut = cut0 + i*0.1
    lmp.set_variable("cut",cut)              # set a variable in LAMMPS
    lmp.command("pair_style lj/cut ${cut}")  # LAMMPS commands
    lmp.command("pair_coeff * * 1.0 1.0")
    lmp.command("run 100")
"""

python source funcdef.py

python source here "from lammps import lammps"
```

1.83.3 Description

The *python* command allows interfacing LAMMPS with an embedded Python interpreter and enables either executing arbitrary python code in that interpreter, registering a Python function for future execution (as a python style variable, from a fix interfaced with python, or for direct invocation), or invoking such a previously registered function.

Arguments, including LAMMPS variables, can be passed to the function from the LAMMPS input script and a value returned by the Python function assigned to a LAMMPS variable. The Python code for the function can be included directly in the input script or in a separate Python file. The function can be standard Python code or it can make "callbacks" to LAMMPS through its library interface to query or set internal values within LAMMPS. This is a powerful mechanism for performing complex operations in a LAMMPS input script that are not possible with the simple input script and variable syntax which LAMMPS defines. Thus your input script can operate more like a true programming language.

Use of this command requires building LAMMPS with the PYTHON package which links to the Python library so that the Python interpreter is embedded in LAMMPS. More details about this process are given below.

There are two ways to invoke a Python function once it has been registered. One is using the *invoke* keyword. The other is to assign the function to a *python-style variable* defined in your input script. Whenever the variable is evaluated, it will execute the Python function to assign a value to the variable. Note that variables can be evaluated in many different ways within LAMMPS. They can be substituted with their result directly in an input script, or they can be passed to various commands as arguments, so that the variable is evaluated during a simulation run.

A broader overview of how Python can be used with LAMMPS is given in the *Use Python with LAMMPS* section of

the documentation. There also is an examples/python directory which illustrates use of the python command.

The first argument of the *python* command is either the *source* keyword or the name of a Python function. This defines the mode of the python command.

Changed in version 22Dec2022.

If the *source* keyword is used, it is followed by either a file name or the *here* keyword. No other keywords can be used. The *here* keyword is followed by a string with python commands, either on a single line enclosed in quotes, or as multiple lines enclosed in triple quotes. These Python commands will be passed to the python interpreter and executed immediately without registering a Python function for future execution. The code will be loaded into and run in the "main" module of the Python interpreter. This allows running arbitrary Python code at any time while processing the LAMMPS input file. This can be used to pre-load Python modules, initialize global variables, define functions or classes, or perform operations using the python programming language. The Python code will be executed in parallel on all MPI processes. No arguments can be passed.

In all other cases, the first argument is the name of a Python function that will be registered with LAMMPS for future execution. The function may already be defined (see *exists* keyword) or must be defined using the *file* or *here* keywords as explained below.

If the *invoke* keyword is used, no other keywords can be used, and a previous *python* command must have registered the Python function referenced by this command. This invokes the Python function with the previously defined arguments and the return value is processed as explained below. You can invoke the function as many times as you wish in your input script.

The *input* keyword defines how many arguments *N* the Python function expects. If it takes no arguments, then the *input* keyword should not be used. Each argument can be specified directly as a value, e.g. '6' or '3.14159' or 'abc' (a string of characters). The type of each argument is specified by the *format* keyword as explained below, so that Python will know how to interpret the value. If the word SELF is used for an argument it has a special meaning. A pointer is passed to the Python function which it can convert into a reference to LAMMPS itself using the *LAMMPS Python module*. This enables the function to call back to LAMMPS through its library interface as explained below. This allows the Python function to query or set values internal to LAMMPS which can affect the subsequent execution of the input script. A LAMMPS variable can also be used as an argument, specified as v_name, where "name" is the name of the variable. Any style of LAMMPS variable returning a scalar or a string can be used, as defined by the *variable* command. The *format* keyword must be used to set the type of data that is passed to Python. Each time the Python function is invoked, the LAMMPS variable is evaluated and its value is passed to the Python function.

The *return* keyword is only needed if the Python function returns a value. The specified *varReturn* must be of the form v_name, where "name" is the name of a python-style LAMMPS variable, defined by the *variable* command. The Python function can return a numeric or string value, as specified by the *format* keyword.

As explained on the *variable* doc page, the definition of a python-style variable associates a Python function name with the variable. This must match the *Python function name* first argument of the *python* command. For example these two commands would be consistent:

```
variable foo python myMultiply
python myMultiply return v_foo format f file funcs.py
```

The two commands can appear in either order in the input script so long as both are specified before the Python function is invoked for the first time. Afterwards, the variable 'foo' is associated with the Python function 'myMultiply'.

The *format* keyword must be used if the *input* or *return* keywords are used. It defines an *fstring* with M characters, where M = sum of number of inputs and outputs. The order of characters corresponds to the N inputs, followed by the return value (if it exists). Each character must be one of the following: "i" for integer, "f" for floating point, "s" for string, or "p" for SELF. Each character defines the type of the corresponding input or output value of the Python function and affects the type conversion that is performed internally as data is passed back and forth between LAMMPS

and Python. Note that it is permissible to use a *python-style variable* in a LAMMPS command that allows for an equal-style variable as an argument, but only if the output of the Python function is flagged as a numeric value ("i" or "f") via the *format* keyword.

If the *return* keyword is used and the *format* keyword specifies the output as a string, then the default maximum length of that string is 63 characters (64-1 for the string terminator). If you want to return a longer string, the *length* keyword can be specified with its *Nlen* value set to a larger number (the code allocates space for Nlen+1 to include the string terminator). If the Python function generates a string longer than the default 63 or the specified *Nlen*, it will be truncated.

Either the *file*, *here*, or *exists* keyword must be used, but only one of them. These keywords specify what Python code to load into the Python interpreter. The *file* keyword gives the name of a file containing Python code, which should end with a ".py" suffix. The code will be immediately loaded into and run in the "main" module of the Python interpreter. The Python code will be executed in parallel on all MPI processes. Note that Python code which contains a function definition does not "execute" the function when it is run; it simply defines the function so that it can be invoked later.

The *here* keyword does the same thing, except that the Python code follows as a single argument to the *here* keyword. This can be done using triple quotes as delimiters, as in the examples above. This allows Python code to be listed verbatim in your input script, with proper indentation, blank lines, and comments, as desired. See the *Commands parse* doc page, for an explanation of how triple quotes can be used as part of input script syntax.

The *exists* keyword takes no argument. It means that Python code containing the required Python function with the given name has already been executed, for example by a *python source* command or in the same file that was used previously with the *file* keyword.

Note that the Python code that is loaded and run must contain a function with the specified function name. To operate properly when later invoked, the function code must match the *input* and *return* and *format* keywords specified by the python command. Otherwise Python will generate an error.

This section describes how Python code can be written to work with LAMMPS.

Whether you load Python code from a file or directly from your input script, via the *file* and *here* keywords, the code can be identical. It must be indented properly as Python requires. It can contain comments or blank lines. If the code is in your input script, it cannot however contain triple-quoted Python strings, since that will conflict with the triple-quote parsing that the LAMMPS input script performs.

All the Python code you specify via one or more python commands is loaded into the Python "main" module, i.e. __name__ == '__main__'. The code can define global variables, define global functions, define classes or execute statements that are outside of function definitions. It can contain multiple functions, only one of which matches the *func* setting in the python command. This means you can use the *file* keyword once to load several functions, and the *exists* keyword thereafter in subsequent python commands to register the other functions that were previously loaded with LAMMPS.

A Python function you define (or more generally, the code you load) can import other Python modules or classes, it can make calls to other system functions or functions you define, and it can access or modify global variables (in the "main" module) which will persist between successive function calls. The latter can be useful, for example, to prevent a function from being invoke multiple times per timestep by different commands in a LAMMPS input script that access the returned python-style variable associated with the function. For example, consider this function loaded with two global variables defined outside the function:

```
nsteplast = -1
nvaluelast = 0

def expensive(nstep):
  global nsteplast,nvaluelast
  if nstep == nsteplast: return nvaluelast
```

(continues on next page)

(continued from previous page)

```
  nsteplast = nstep
  # perform complicated calculation
  nvalue = ...
  nvaluelast = nvalue
  return nvalue
```

The variable 'nsteplast' stores the previous timestep the function was invoked (passed as an argument to the function). The variable 'nvaluelast' stores the return value computed on the last function invocation. If the function is invoked again on the same timestep, the previous value is simply returned, without re-computing it. The "global" statement inside the Python function allows it to overwrite the global variables from within the local context of the function.

Note that if you load Python code multiple times (via multiple python commands), you can overwrite previously loaded variables and functions if you are not careful. E.g. if the code above were loaded twice, the global variables would be re-initialized, which might not be what you want. Likewise, if a function with the same name exists in two chunks of Python code you load, the function loaded second will override the function loaded first.

It's important to realize that if you are running LAMMPS in parallel, each MPI task will load the Python interpreter and execute a local copy of the Python function(s) you define. There is no connection between the Python interpreters running on different processors. This implies three important things.

First, if you put a print or other statement creating output to the screen in your Python function, you will see P copies of the output, when running on P processors. If the prints occur at (nearly) the same time, the P copies of the output may be mixed together. When loading the LAMMPS Python module into the embedded Python interpreter, it is possible to pass the pointer to the current LAMMPS class instance and via the Python interface to the LAMMPS library interface, it is possible to determine the MPI rank of the current process and thus adapt the Python code so that output will only appear on MPI rank 0. The following LAMMPS input demonstrates how this could be done. The text 'Hello, LAMMPS!' should be printed only once, even when running LAMMPS in parallel.

```
python python_hello input 1 SELF format p here """
def python_hello(handle):
    from lammps import lammps
    lmp = lammps(ptr=handle)
    me = lmp.extract_setting('world_rank')
    if me == 0:
        print('Hello, LAMMPS!')
"""

python python_hello invoke
```

If your Python code loads Python modules that are not pre-loaded by the Python library, then it will load the module from disk. This may be a bottleneck if 1000s of processors try to load a module at the same time. On some large supercomputers, loading of modules from disk by Python may be disabled. In this case you would need to pre-build a Python library that has the required modules pre-loaded and link LAMMPS with that library.

Third, if your Python code calls back to LAMMPS (discussed in the next section) and causes LAMMPS to perform an MPI operation requires global communication (e.g. via MPI_Allreduce), such as computing the global temperature of the system, then you must ensure all your Python functions (running independently on different processors) call back to LAMMPS. Otherwise the code may hang.

Your Python function can "call back" to LAMMPS through its library interface, if you use the SELF input to pass Python a pointer to LAMMPS. The mechanism for doing this in your Python function is as follows:

```
def foo(handle,...):
  from lammps import lammps
  lmp = lammps(ptr=handle)
  lmp.command('print "Hello from inside Python"')
  ...
```

The function definition must include a variable ('handle' in this case) which corresponds to SELF in the *python* command. The first line of the function imports the *"lammps" Python module*. The second line creates a Python object lmp which wraps the instance of LAMMPS that called the function. The 'ptr=handle' argument is what makes that happen. The third line invokes the command() function in the LAMMPS library interface. It takes a single string argument which is a LAMMPS input script command for LAMMPS to execute, the same as if it appeared in your input script. In this case, LAMMPS should output

```
Hello from inside Python
```

to the screen and log file. Note that since the LAMMPS print command itself takes a string in quotes as its argument, the Python string must be delimited with a different style of quotes.

The *Use Python with LAMMPS* page describes the syntax for how Python wraps the various functions included in the LAMMPS library interface.

A more interesting example is in the examples/python/in.python script which loads and runs the following function from examples/python/funcs.py:

```
def loop(N,cut0,thresh,lmpptr):
  print("LOOP ARGS", N, cut0, thresh, lmpptr)
  from lammps import lammps
  lmp = lammps(ptr=lmpptr)
  natoms = lmp.get_natoms()

  for i in range(N):
    cut = cut0 + i*0.1

    lmp.set_variable("cut",cut)               # set a variable in LAMMPS
    lmp.command("pair_style lj/cut ${cut}")   # LAMMPS command
    #lmp.command("pair_style lj/cut %d" % cut)  # LAMMPS command option

    lmp.command("pair_coeff * * 1.0 1.0")     # ditto
    lmp.command("run 10")                     # ditto
    pe = lmp.extract_compute("thermo_pe",0,0)  # extract total PE from LAMMPS
    print("PE", pe/natoms, thresh)
    if pe/natoms < thresh: return
```

with these input script commands:

```
python        loop input 4 10 1.0 -4.0 SELF format iffp file funcs.py
python        loop invoke
```

This has the effect of looping over a series of 10 short runs (10 timesteps each) where the pair style cutoff is increased from a value of 1.0 in distance units, in increments of 0.1. The looping stops when the per-atom potential energy falls below a threshold of -4.0 in energy units. More generally, Python can be used to implement a loop with complex logic, much more so than can be created using the LAMMPS *jump* and *if* commands.

Several LAMMPS library functions are called from the loop function. Get_natoms() returns the number of atoms in the simulation, so that it can be used to normalize the potential energy that is returned by extract_compute() for the "thermo_pe" compute that is defined by default for LAMMPS thermodynamic output. Set_variable() sets the value of

a string variable defined in LAMMPS. This library function is a useful way for a Python function to return multiple values to LAMMPS, more than the single value that can be passed back via a return statement. This cutoff value in the "cut" variable is then substituted (by LAMMPS) in the pair_style command that is executed next. Alternatively, the "LAMMPS command option" line could be used in place of the 2 preceding lines, to have Python insert the value into the LAMMPS command string.

Note

When using the callback mechanism just described, recognize that there are some operations you should not attempt because LAMMPS cannot execute them correctly. If the Python function is invoked between runs in the LAMMPS input script, then it should be OK to invoke any LAMMPS input script command via the library interface command() or file() functions, so long as the command would work if it were executed in the LAMMPS input script directly at the same point.

However, a Python function can also be invoked during a run, whenever an associated LAMMPS variable it is assigned to is evaluated. If the variable is an input argument to another LAMMPS command (e.g. *fix setforce*), then the Python function will be invoked inside the class for that command, in one of its methods that is invoked in the middle of a timestep. You cannot execute arbitrary input script commands from the Python function (again, via the command() or file() functions) at that point in the run and expect it to work. Other library functions such as those that invoke computes or other variables may have hidden side effects as well. In these cases, LAMMPS has no simple way to check that something illogical is being attempted.

The same applies to Python functions called during a simulation run at each time step using *fix python/invoke*.

If you run Python code directly on your workstation, either interactively or by using Python to launch a Python script stored in a file, and your code has an error, you will typically see informative error messages. That is not the case when you run Python code from LAMMPS using an embedded Python interpreter. The code will typically fail silently. LAMMPS will catch some errors but cannot tell you where in the Python code the problem occurred. For example, if the Python code cannot be loaded and run because it has syntax or other logic errors, you may get an error from Python pointing to the offending line, or you may get one of these generic errors from LAMMPS:

```
Could not process Python file
Could not process Python string
```

When the Python function is invoked, if it does not return properly, you will typically get this generic error from LAMMPS:

```
Python function evaluation failed
```

Here are three suggestions for debugging your Python code while running it under LAMMPS.

First, don't run it under LAMMPS, at least to start with! Debug it using plain Python. Load and invoke your function, pass it arguments, check return values, etc.

Second, add Python print statements to the function to check how far it gets and intermediate values it calculates. See the discussion above about printing from Python when running in parallel.

Third, use Python exception handling. For example, say this statement in your Python function is failing, because you have not initialized the variable foo:

```
foo += 1
```

If you put one (or more) statements inside a "try" statement, like this:

```
import exceptions
print("Inside simple function")
try:
  foo += 1      # one or more statements here
except Exception as e:
  print("FOO error:", e)
```

then you will get this message printed to the screen:

```
FOO error: local variable 'foo' referenced before assignment
```

If there is no error in the try statements, then nothing is printed. Either way the function continues on (unless you put a return or sys.exit() in the except clause).

1.83.4 Restrictions

This command is part of the PYTHON package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Building LAMMPS with the PYTHON package will link LAMMPS with the Python library on your system. Settings to enable this are in the lib/python/Makefile.lammps file. See the lib/python/README file for information on those settings.

If you use Python code which calls back to LAMMPS, via the SELF input argument explained above, there is an extra step required when building LAMMPS. LAMMPS must also be built as a shared library and your Python function must be able to load the *"lammps" Python module* that wraps the LAMMPS library interface. These are the same steps required to use Python by itself to wrap LAMMPS. Details on these steps are explained on the *Python* doc page. Note that it is important that the stand-alone LAMMPS executable and the LAMMPS shared library be consistent (built from the same source code files) in order for this to work. If the two have been built at different times using different source files, problems may occur.

Another limitation of calling back to Python from the LAMMPS module using the *python* command in a LAMMPS input is that both, the Python interpreter and LAMMPS, must be linked to the same Python runtime as a shared library. If the Python interpreter is linked to Python statically (which seems to happen with Conda) then loading the shared LAMMPS library will create a second python "main" module that hides the one from the Python interpreter and all previous defined function and global variables will become invisible.

1.83.5 Related commands

shell, *variable*, *fix python/invoke*

1.83.6 Default

none

1.84 quit command

1.84.1 Syntax

```
quit status
```

status = numerical exit status (optional)

1.84.2 Examples

```
quit
if "$n > 10000" then "quit 1"
```

1.84.3 Description

This command causes LAMMPS to exit, after shutting down all output cleanly.

It can be used as a debug statement in an input script, to terminate the script at some intermediate point.

It can also be used as an invoked command inside the "then" or "else" portion of an *if* command.

The optional status argument is an integer which signals the return status to a program calling LAMMPS. A return status of 0 usually indicates success. A status != 0 is failure, where the specified value can be used to distinguish the kind of error, e.g. where in the input script the quit was invoked. If not specified, a status of 0 is returned.

1.84.4 Restrictions

none

1.84.5 Related commands

if

1.84.6 Default

none

1.85 read_data command

1.85.1 Syntax

```
read_data file keyword args ...
```

- file = name of data file to read in
- zero or more keyword/arg pairs may be appended

- keyword = *add* or *offset* or *shift* or *extra/atom/types* or *extra/bond/types* or *extra/angle/types* or *extra/dihedral/types* or *extra/improper/types* or *extra/bond/per/atom* or *extra/angle/per/atom* or *extra/dihedral/per/atom* or *extra/improper/per/atom* or *extra/special/per/atom* or *group* or *nocoeff* or *fix*

  add arg = append or IDoffset or IDoffset MOLoffset or merge
    append = add new atoms with atom IDs appended to current IDs
    IDoffset = add new atoms with atom IDs having IDoffset added
    MOLoffset = add new atoms with molecule IDs having MOLoffset added (only when molecule IDs are enabled)
    merge = add new atoms with their atom IDs (and molecule IDs) unchanged
  offset args = toff boff aoff doff ioff
    toff = offset to add to atom types
    boff = offset to add to bond types
    aoff = offset to add to angle types
    doff = offset to add to dihedral types
    ioff = offset to add to improper types
  shift args = Sx Sy Sz
    Sx,Sy,Sz = distance to shift atoms when adding to system (distance units)
  extra/atom/types arg = # of extra atom types
  extra/bond/types arg = # of extra bond types
  extra/angle/types arg = # of extra angle types
  extra/dihedral/types arg = # of extra dihedral types
  extra/improper/types arg = # of extra improper types
  extra/bond/per/atom arg = leave space for this many new bonds per atom
  extra/angle/per/atom arg = leave space for this many new angles per atom
  extra/dihedral/per/atom arg = leave space for this many new dihedrals per atom
  extra/improper/per/atom arg = leave space for this many new impropers per atom
  extra/special/per/atom arg = leave space for extra 1-2,1-3,1-4 interactions per atom
  group args = groupID
    groupID = add atoms in data file to this group
  nocoeff = ignore force field parameters
  fix args = fix-ID header-string section-string
    fix-ID = ID of fix to process header lines and sections of data file
    header-string = header lines containing this string will be passed to fix
    section-string = section names with this string will be passed to fix

1.85.2 Examples

```
read_data data.lj
read_data ../run7/data.polymer.gz
read_data data.protein fix mycmap crossterm CMAP
read_data data.water add append offset 3 1 1 1 1 shift 0.0 0.0 50.0
read_data data.water add merge group solvent
```

1.85.3 Description

Read in a data file containing information LAMMPS needs to run a simulation. The file can be ASCII text or a gzipped text file (detected by a .gz suffix).

This is one of 3 ways to specify the simulation box: see the *create_box* and *read_restart* and commands for alternative methods. It is also one of 3 ways to specify initial atom coordinates: see the *create_atoms* and *read_restart* and commands for alternative methods. Also see the explanation of the *-restart command-line switch* which can convert a restart file to a data file.

This command can be used multiple times to add new atoms and their properties to an existing system by using the *add*, *offset*, and *shift* keywords. However, it is important to understand that several system parameters, like the number of types of different kinds and per atom settings are **locked in** after the first *read_data* command, which means that no type ID (including its offset) may have a larger value when processing additional data files than what is set by the first data file and the corresponding *read_data* command options. See more details on this situation below, which includes the use case for the *extra* keywords.

The *group* keyword adds all the atoms in the data file to the specified group-ID. The group will be created if it does not already exist. This is useful if you are reading multiple data files and wish to put sets of atoms into different groups so they can be operated on later. E.g. a group of added atoms can be moved to new positions via the *displace_atoms* command. Note that atoms read from the data file are also always added to the "all" group. The *group* command discusses atom groups, as used in LAMMPS.

The *nocoeff* keyword tells read_data to ignore force field parameters. The various Coeff sections are still read and have to have the correct number of lines, but they are not applied. This also allows to read a data file without having any pair, bond, angle, dihedral or improper styles defined, or to read a data file for a different force field.

The use of the *fix* keyword is discussed below.

1.85.4 Reading multiple data files

The read_data command can be used multiple times with the same or different data files to build up a complex system from components contained in individual data files. For example one data file could contain fluid in a confined domain; a second could contain wall atoms, and the second file could be read a third time to create a wall on the other side of the fluid. The third set of atoms could be rotated to an opposing direction using the *displace_atoms* command, after the third read_data command is used.

The *add*, *offset*, *shift*, *extra*, and *group* keywords are useful in this context.

If a simulation box does not yet exist, the *add* keyword cannot be used; the read_data command is being used for the first time. If a simulation box does exist, due to using the *create_box* command, or a previous read_data command, then the *add* keyword must be used.

Note

If the first read_data command defined an orthogonal or restricted triclinic or general triclinic simulation box (see the sub-section below on header keywords), then subsequent data files must define the same kind of simulation box. For orthogonal boxes, the new box can be a different size; see the next Note. For a restricted triclinic box, the 3 new tilt factors ("xy xz yz" keyword) must have the same values as in the original data file. For a general triclinic box, the new avec, bvec, cvec, and "abc origin" keywords must have the same values in the original data file. files. Also the *shift* keyword cannot be used in subsequent read_data commands for a general triclinic box.

Note

For orthogonal boxes, the simulation box size in the new data file will be merged with the existing simulation box to create a large enough box in each dimension to contain both the existing and new atoms. Each box dimension never shrinks due to this merge operation, it only stays the same or grows. Care must be used if you are growing the existing simulation box in a periodic dimension. If there are existing atoms with bonds that straddle that periodic boundary, then the atoms may become far apart if the box size grows. This will separate the atoms in the bond, which can lead to "lost" bond atoms or bad dynamics.

The three choices for the *add* argument affect how the atom IDs and molecule IDs of atoms in the data file are treated.

If *append* is specified, atoms in the data file are added to the current system, with their atom IDs reset so that an atom-ID = M in the data file becomes atom-ID = N+M, where N is the largest atom ID in the current system. This rule is applied to all occurrences of atom IDs in the data file, e.g. in the Velocity or Bonds section. This is also done for molecule IDs, if the atom style does support molecule IDs or they are enabled via fix property/atom.

If *IDoffset* is specified, then *IDoffset* is a numeric value is given, e.g. 1000, so that an atom-ID = M in the data file becomes atom-ID = 1000+M. For systems with enabled molecule IDs, another numerical argument *MOLoffset* is required representing the equivalent offset for molecule IDs.

If *merge* is specified, the data file atoms are added to the current system without changing their IDs. They are assumed to merge (without duplication) with the currently defined atoms. It is up to you to ensure there are no multiply defined atom IDs, as LAMMPS only performs an incomplete check that this is the case by ensuring the resulting max atom-ID >= the number of atoms. For molecule IDs, there is no check done at all.

The *offset* and *shift* keywords can only be used if the *add* keyword is also specified.

The *offset* keyword adds the specified offset values to the atom types, bond types, angle types, dihedral types, and improper types as they are read from the data file. E.g. if *toff* = 2, and the file uses atom types 1,2,3, then the added atoms will have atom types 3,4,5. These offsets apply to all occurrences of types in the data file, e.g. for the Atoms or Masses or Pair Coeffs or Bond Coeffs sections. This makes it easy to use atoms and molecules and their attributes from a data file in different simulations, where you want their types (atom, bond, angle, etc) to be different depending on what other types already exist. All five offset values must be specified, but individual values will be ignored if the data file does not use that attribute (e.g. no bonds).

Note

Offsets are **ignored** on lines using type labels, as the type labels will determine the actual types directly depending on the current *labelmap* settings.

The *shift* keyword can be used to specify an (Sx, Sy, Sz) displacement applied to the coordinates of each atom. Sz must be 0.0 for a 2d simulation. This is a mechanism for adding structured collections of atoms at different locations within the simulation box, to build up a complex geometry. It is up to you to ensure atoms do not end up overlapping unphysically which would lead to bad dynamics. Note that the *displace_atoms* command can be used to move a subset of atoms after they have been read from a data file. Likewise, the *delete_atoms* command can be used to remove overlapping atoms. Note that the shift values (Sx, Sy, Sz) are also added to the simulation box information (xlo, xhi, ylo, yhi, zlo, zhi) in the data file to shift its boundaries. E.g. xlo_new = xlo + Sx, xhi_new = xhi + Sx.

The *extra* keywords can only be used the first time the read_data command is used. They are useful if you intend to add new atom, bond, angle, etc types later with additional read_data commands. This is because the maximum number of allowed atom, bond, angle, etc types is set by LAMMPS when the system is first initialized. If you do not use the *extra* keywords, then the number of these types will be limited to what appears in the first data file you read. For example, if the first data file is a solid substrate of Si, it will likely specify a single atom type. If you read a second data file with a different material (water molecules) that sit on top of the substrate, you will want to use different atom types for those atoms. You can only do this if you set the *extra/atom/types* keyword to a sufficiently large value when reading

the substrate data file. Note that use of the *extra* keywords also allows each data file to contain sections like Masses or Pair Coeffs or Bond Coeffs which are sized appropriately for the number of types in that data file. If the *offset* keyword is used appropriately when each data file is read, the values in those sections will be stored correctly in the larger data structures allocated by the use of the *extra* keywords. E.g. the substrate file can list mass and pair coefficients for type 1 silicon atoms. The water file can list mass and pair coefficients for type 1 and type 2 hydrogen and oxygen atoms. Use of the *extra* and *offset* keywords will store those mass and pair coefficient values appropriately in data structures that allow for 3 atom types (Si, H, O). Of course, you would still need to specify coefficients for H/Si and O/Si interactions in your input script to have a complete pairwise interaction model.

An alternative to using the *extra* keywords with the read_data command, is to use the *create_box* command to initialize the simulation box and all the various type limits you need via its *extra* keywords. Then use the read_data command one or more times to populate the system with atoms, bonds, angles, etc, using the *offset* keyword if desired to alter types used in the various data files you read.

1.85.5 Format of a data file

The structure of the data file is important, though many settings and sections are optional or can come in any order. See the examples directory for sample data files for different problems.

The file will be read line by line, but there is a limit of 254 characters per line and characters beyond that limit will be ignored.

A data file has a header and a body. The header appears first. The first line of the header and thus of the data file is *always* skipped; it typically contains a description of the file or a comment from the software that created the file.

Then lines are read one line at a time. Lines can have a trailing comment starting with '#' that is ignored. There *must* be at least one blank between any valid content and the comment. If a line is blank (i.e. contains only white-space after comments are deleted), it is skipped. If the line contains a header keyword, the corresponding value(s) is/are read from the line. A line that is *not* blank and does *not* contain a header keyword begins the body of the file.

The body of the file contains zero or more sections. The first line of a section has only a keyword. This line can have a trailing comment starting with '#' that is either ignored or can be used to check for a style match, as described below. There must be a blank between the keyword and any comment. The *next* line is *always* skipped. The remaining lines of the section contain values. The number of lines depends on the section keyword as described below. Zero or more blank lines can be used *between* sections. Sections can appear in any order, with a few exceptions as noted below.

The keyword *fix* can be used one or more times. Each usage specifies a fix that will be used to process a specific portion of the data file. Any header line containing *header-string* and any section that is an exact match to *section-string* will be passed to the specified fix. See the *fix property/atom* command for an example of a fix that operates in this manner. The doc page for the fix defines the syntax of the header line(s) and section that it reads from the data file. Note that the *header-string* can be specified as NULL, in which case no header lines are passed to the fix. This means the fix can infer the length of its Section from standard header settings, such as the number of atoms. Also the *section-string* may be specified as NULL, and in that case the fix ID is used as section name.

The formatting of individual lines in the data file (indentation, spacing between words and numbers) is not important except that header and section keywords (e.g. atoms, xlo xhi, Masses, Bond Coeffs) must be capitalized as shown and cannot have extra white-space between their words - e.g. two spaces or a tab between the 2 words in "xlo xhi" or the 2 words in "Bond Coeffs", is not valid.

1.85.6 Format of the header of a data file

These are the recognized header keywords. Header lines can come in any order. Each keyword takes a single value unless noted in this list. The value(s) are read from the beginning of the line. Thus the keyword *atoms* should be in a line like "1000 atoms"; the keyword *ylo yhi* should be in a line like "-10.0 10.0 ylo yhi"; the keyword *xy xz yz* should be in a line like "0.0 5.0 6.0 xy xz yz".

All these settings have a default value of 0, except for the simulation box size settings; their defaults are explained below. A keyword line need only appear if its value is different than the default.

- *atoms* = # of atoms in system
- *bonds* = # of bonds in system
- *angles* = # of angles in system
- *dihedrals* = # of dihedrals in system
- *impropers* = # of impropers in system
- *atom types* = # of atom types in system
- *bond types* = # of bond types in system
- *angle types* = # of angle types in system
- *dihedral types* = # of dihedral types in system
- *improper types* = # of improper types in system
- *extra bond per atom* = leave space for this many new bonds per atom (deprecated, use extra/bond/per/atom keyword)
- *extra angle per atom* = leave space for this many new angles per atom (deprecated, use extra/angle/per/atom keyword)
- *extra dihedral per atom* = leave space for this many new dihedrals per atom (deprecated, use extra/dihedral/per/atom keyword)
- *extra improper per atom* = leave space for this many new impropers per atom (deprecated, use extra/improper/per/atom keyword)
- *extra special per atom* = leave space for this many new special bonds per atom (deprecated, use extra/special/per/atom keyword)
- *ellipsoids* = # of ellipsoids in system
- *lines* = # of line segments in system
- *triangles* = # of triangles in system
- *bodies* = # of bodies in system
- *xlo xhi* = simulation box boundaries in x dimension (2 values)
- *ylo yhi* = simulation box boundaries in y dimension (2 values)
- *zlo zhi* = simulation box boundaries in z dimension (2 values)
- *xy xz yz* = simulation box tilt factors for triclinic system (3 values)
- *avec* = first edge vector of a general triclinic simulation box (3 values)
- *bvec* = second edge vector of a general triclinic simulation box (3 values)
- *cvec* = third edge vector of a general triclinic simulation box (3 values)
- *abc origin* = origin of a general triclinic simulation box (3 values)

1.85.7 Header specification of the simulation box size and shape

The last 8 keywords in the list of header keywords are for simulation boxes of 3 kinds which LAMMPS supports:

- orthogonal box = faces are perpendicular to the xyz coordinate axes
- restricted triclinic box = a parallelepiped defined by 3 edge vectors oriented in a constrained manner
- general triclinic box = a parallelepiped defined by 3 arbitrary edge vectors

For restricted and general triclinic boxes, see the *Howto_triclinic* doc page for a fuller description than is given here.

The units of the values for all 8 keywords in in distance units; see the *units* command for details.

For all 3 kinds of simulation boxes, the system may be periodic or non-periodic in any dimension; see the *boundary* command for details.

When the simulation box is created by the read_data command, it is also partitioned into a regular 3d grid of subdomains, one per processor, based on the number of processors being used and the settings of the *processors* command. For each kind of simulation box the subdomains have the same shape as the simulation box, i.e. smaller orthogonal bricks for orthogonal boxes, smaller parallelepipeds for triclinic boxes. The partitioning can later be changed by the *balance* or *fix balance* commands.

For an orthogonal box, only the *xlo xhi*, *ylo yhi*, *zlo zhi* keywords are used. They define the extent of the simulation box in each dimension so that the resulting edge vectors of an orthogonal box are:

- **A** = (xhi-xlo,0,0)
- **B** = (0,yhi-ylo,0)
- **C** = (0,0,zhi-zlo)

The origin (lower left corner) of the orthogonal box is at (xlo,ylo,zlo). The default values for these 3 keywords are -0.5 and 0.5 for each lo/hi pair. For a 2d simulation, the zlo and zhi values must straddle zero. The default zlo/zhi values do this, so that keyword is not needed in 2d.

For a restricted triclinic box, the *xy xz yz* keyword is used in addition to the *xlo xhi*, *ylo yhi*, *zlo zhi* keywords. The three *xy,xz,yz* values can be 0.0 or positive or negative, and are called "tilt factors" because they are the amount of displacement applied to edges of faces of an orthogonal box to transform it into a restricted triclinic parallelepiped.

The *Howto_triclinic* doc page discusses the tilt factors in detail and explains that the resulting edge vectors of a restricted triclinic box are:

- **A** = (xhi-xlo,0,0)
- **B** = (xy,yhi-ylo,0)
- **C** = (xz,yz,zhi-zlo)

This restricted form of edge vectors requires that **A** be in the direction of the x-axis, **B** be in the xy plane with its y-component in the +y direction, and **C** have its z-component in the +z direction. The origin (lower left corner) of the restricted triclinic box is at (xlo,ylo,zlo).

For a 2d simulation, the zlo and zhi values must straddle zero. The default zlo/zhi values do this, so that keyword is not needed in 2d. The xz and yz values must also be zero in 2d. The shape of the 2d restricted triclinic simulation box is effectively a parallelogram.

Note

When a restricted triclinic box is used, the simulation domain should normally be periodic in any dimensions that tilt is applied to, which is given by the second dimension of the tilt factor (e.g. y for xy tilt). This is so that pairs of atoms interacting across that boundary will have one of them shifted by the tilt factor. Periodicity is set by the *boundary* command which also describes the shifting by the tilt factor. For example, if the xy tilt factor is non-zero, then the y dimension should be periodic. Similarly, the z dimension should be periodic if xz or yz is non-zero. LAMMPS does not require this periodicity, but you may lose atoms if this is not the case.

Note

Normally, the specified tilt factors (xy,xz,yz) should not skew the simulation box by more than half the distance of the corresponding parallel box length for computational efficiency. For example, if $x_{\rm lo} = 2$ and $x_{\rm hi} = 12$, then the x box length is 10 and the xy tilt factor should be between -5 and 5. LAMMPS will issue a warning if this is not the case. See the last sub-section of the *Howto_triclinic* doc page for more details.

Note

If a simulation box is initially orthogonal, but will tilt during a simulation, e.g. via the *fix deform* command, then the box should be defined as restricted triclinic with all 3 tilt factors = 0.0. Alternatively, the *change box* command can be used to convert an orthogonal box to a restricted triclinic box.

For a general triclinic box, the *avec*, *bvec*, *cvec*, and *abc origin* keywords are used. The *xlo xhi*, *ylo yhi*, *zlo zhi*, and *xy xz yz* keywords are NOT used. The first 3 keywords define the 3 edge vectors **A**, **B**, **C** of the general triclinic box. They can be arbitrary vectors so long as they are distinct, non-zero, and not co-planar. They must also define a right-handed system such that (**A** x **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 origin of the box (origin of the 3 edge vectors) is set by the *abc origin* keyword.

The default values for these 4 keywords are as follows:

- avec = (1,0,0)
- bvec = (0,1,0)
- cvec = (0,0,1)
- abc origin = (0,0,0) for 3d, (0,0,-0.5) for 2d

For 2d simulations, *cvec* = (0,0,1) is required, and the 3rd value of *abc origin* must be -0.5. These are the default values, so the *cvec* keyword is not needed in 2d.

Note

LAMMPS allows specification of general triclinic simulation boxes as a convenience for users who may be converting data from solid-state crystallographic representations or from DFT codes for input to LAMMPS. However, as explained on the *Howto_triclinic* doc page, internally, LAMMPS only uses restricted triclinic simulation boxes. This means the box and per-atom information (e.g. coordinates, velocities) in the data file are converted (rotated) from general to restricted triclinic form when the file is read. Other sections of the data file must also list their per-atom data appropriately if vector quantities are specified. This requirement is explained below for the relevant sections. The *Howto_triclinic* doc page also discusses other LAMMPS commands which can input/output general

triclinic representations of the simulation box and per-atom data.

The following explanations apply to all 3 kinds of simulation boxes: orthogonal, restricted triclinic, and general triclinic.

If the system is periodic (in a dimension), then atom coordinates can be outside the bounds (in that dimension); they will be remapped (in a periodic sense) back inside the box. For triclinic boxes, periodicity in x,y,z refers to the faces of the parallelepiped defined by the **A**,**B**,**C** edge vectors of the simulation box. See the *boundary* command doc page for a fuller discussion.

Note that if the *add* option is being used to add atoms to a simulation box that already exists, this periodic remapping will be performed using simulation box bounds that are the union of the existing box and the box boundaries in the new data file.

If the system is non-periodic (in a dimension), then an image flag for that direction has no meaning, since there cannot be periodic images without periodicity and the data file is therefore - technically speaking - invalid. This situation would happen when a data file was written with periodic boundaries and then read back for non-periodic boundaries. Accepting a non-zero image flag can lead to unexpected results for any operations and computations in LAMMPS that internally use unwrapped coordinates (for example computing the center of mass of a group of atoms). Thus all non-zero image flags for non-periodic dimensions will be be reset to zero on reading the data file and LAMMPS will print a warning message, if that happens. This is equivalent to wrapping atoms individually back into the principal unit cell in that direction. This operation is equivalent to the behavior of the *change_box command* when used to change periodicity.

If those atoms with non-zero image flags are involved in bonded interactions, this reset can lead to undesired changes, when the image flag values differ between the atoms, i.e. the bonded interaction straddles domain boundaries. For example a bond can become stretched across the unit cell if one of its atoms is wrapped to one side of the cell and the second atom to the other. In those cases the data file needs to be pre-processed externally to become valid again. This can be done by first unwrapping coordinates and then wrapping entire molecules instead of individual atoms back into the principal simulation cell and finally expanding the cell dimensions in the non-periodic direction as needed, so that the image flag would be zero.

Note

If the system is non-periodic (in a dimension), then all atoms in the data file must have coordinates (in that dimension) that are "greater than or equal to" the lo value and "less than or equal to" the hi value. If the non-periodic dimension is of style "fixed" (see the *boundary* command), then the atom coords must be strictly "less than" the hi value, due to the way LAMMPS assign atoms to processors. Note that you should not make the lo/hi values radically smaller/larger than the extent of the atoms. For example, if atoms extend from 0 to 50, you should not specify the box bounds as -10000 and 10000 unless you also use the *processors command*. This is because LAMMPS uses the specified box size to layout the 3d grid of processors. A huge (mostly empty) box will be sub-optimal for performance when using "fixed" boundary conditions (see the *boundary* command). When using "shrink-wrap" boundary conditions (see the *boundary* command), a huge (mostly empty) box may cause a parallel simulation to lose atoms when LAMMPS shrink-wraps the box around the atoms. The read_data command will generate an error in this case.

1.85.8 Meaning of other header keywords

The "extra bond per atom" setting (angle, dihedral, improper) is only needed if new bonds (angles, dihedrals, impropers) will be added to the system when a simulation runs, e.g. by using the *fix bond/create* command. Using this header flag is deprecated; please use the *extra/bond/per/atom* keyword (and correspondingly for angles, dihedrals and impropers) in the read_data command instead. Either will pre-allocate space in LAMMPS data structures for storing the new bonds (angles, dihedrals, impropers).

The "extra special per atom" setting is typically only needed if new bonds/angles/etc will be added to the system, e.g. by using the *fix bond/create* command. Or if entire new molecules will be added to the system, e.g. by using the *fix deposit* or *fix pour* commands, which will have more special 1-2,1-3,1-4 neighbors than any other molecules defined in the data file. Using this header flag is deprecated; please use the *extra/special/per/atom* keyword instead. Using this setting will pre-allocate space in the LAMMPS data structures for storing these neighbors. See the *special_bonds* and *molecule* doc pages for more discussion of 1-2,1-3,1-4 neighbors.

Note

All of the "extra" settings are only applied in the first data file read and when no simulation box has yet been created; as soon as the simulation box is created (and read_data implies that), these settings are *locked* and cannot be changed anymore. Please see the description of the *add* keyword above for reading multiple data files. If they appear in later data files, they are ignored.

The "ellipsoids" and "lines" and "triangles" and "bodies" settings are only used with *atom_style ellipsoid or line or tri or body* and specify how many of the atoms are finite-size ellipsoids or lines or triangles or bodies; the remainder are point particles. See the discussion of ellipsoidflag and the *Ellipsoids* section below. See the discussion of lineflag and the *Lines* section below. See the discussion of triangleflag and the *Triangles* section below. See the discussion of bodyflag and the *Bodies* section below.

Note

For *atom_style template*, the molecular topology (bonds,angles,etc) is contained in the molecule templates read-in by the *molecule* command. This means you cannot set the *bonds*, *angles*, etc header keywords in the data file, nor can you define *Bonds*, *Angles*, etc sections as discussed below. You can set the *bond types*, *angle types*, etc header keywords, though it is not necessary. If specified, they must match the maximum values defined in any of the template molecules.

1.85.9 Format of the body of a data file

These are the section keywords for the body of the file.

- *Atoms, Velocities, Masses, Ellipsoids, Lines, Triangles, Bodies* = atom-property sections
- *Bonds, Angles, Dihedrals, Impropers* = molecular topology sections
- *Atom Type Labels, Bond Type Labels, Angle Type Labels, Dihedral Type Labels, Improper Type Labels* = type label maps
- *Pair Coeffs, PairIJ Coeffs, Bond Coeffs, Angle Coeffs, Dihedral Coeffs, Improper Coeffs* = force field sections
- *BondBond Coeffs, BondAngle Coeffs, MiddleBondTorsion Coeffs, EndBondTorsion Coeffs, AngleTorsion Coeffs, AngleAngleTorsion Coeffs, BondBond13 Coeffs, AngleAngle Coeffs* = class 2 force field sections

These keywords will check an appended comment for a match with the currently defined style:

- *Atoms, Pair Coeffs, PairIJ Coeffs, Bond Coeffs, Angle Coeffs, Dihedral Coeffs, Improper Coeffs*

For example, these lines:

```
Atoms # sphere
Pair Coeffs # lj/cut
```

will check if the currently-defined *atom_style* is *sphere*, and the current *pair_style* is *lj/cut*. If not, LAMMPS will issue a warning to indicate that the data file section likely does not contain the correct number or type of parameters expected for the currently-defined style.

Each section is listed below in alphabetic order. The format of each section is described including the number of lines it must contain and rules (if any) for where it can appear in the data file.

Any individual line in the various sections can have a trailing comment starting with "#" for annotation purposes. There must be at least one blank between valid content and the comment. E.g. in the Atoms section:

```
10 1 17 -1.0 10.0 5.0 6.0   # salt ion
```

Angle Coeffs section:

- one line per angle type
- line syntax: ID coeffs

  ```
  ID = angle type (1-N)
  coeffs = list of coeffs
  ```

- example:

  ```
  6 70 108.5 0 0
  ```

The number and meaning of the coefficients are specific to the defined angle style. See the *angle_style* and *angle_coeff* commands for details. Coefficients can also be set via the *angle_coeff* command in the input script.

Angle Type Labels section:

- one line per angle type
- line syntax: ID label

  ```
  ID = angle type (1-N)
  label = alphanumeric type label
  ```

Define alphanumeric type labels for each numeric angle type. These can be used in the Angles section in place of a numeric type, but only if the this section appears before the Angles section.

See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

AngleAngle Coeffs section:

- one line per improper type
- line syntax: ID coeffs

ID = improper type (1-N)
coeffs = list of coeffs (see improper_coeff)

AngleAngleTorsion Coeffs section:

- one line per dihedral type
- line syntax: ID coeffs

  ID = dihedral type (1-N)
  coeffs = list of coeffs (see dihedral_coeff)

Angles section:

- one line per angle
- line syntax: ID type atom1 atom2 atom3

```
ID = number of angle (1-Nangles)
type = angle type (1-Nangletype, or type label)
atom1,atom2,atom3 = IDs of 1st,2nd,3rd atom in angle
```

example:

```
2 2 17 29 430
```

The three atoms are ordered linearly within the angle. Thus the central atom (around which the angle is computed) is the atom2 in the list. E.g. H,O,H for a water molecule. The *Angles* section must appear after the *Atoms* section.

All values in this section must be integers (1, not 1.0). However, the type can be a numeric value or an alphanumeric label. The latter is only allowed if the type label has been defined by the *labelmap* command or an Angle Type Labels section earlier in the data file. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

AngleTorsion Coeffs section:

- one line per dihedral type
- line syntax: ID coeffs

  ID = dihedral type (1-N)
  coeffs = list of coeffs (see dihedral_coeff)

Atom Type Labels section:

- one line per atom type
- line syntax: ID label

```
ID = numeric atom type (1-N)
label = alphanumeric type label
```

Define alphanumeric type labels for each numeric atom type. These can be used in the Atoms section in place of a numeric type, but only if the Atom Type Labels section appears before the Atoms section.

See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

Atoms section:

- one line per atom
- line syntax: depends on atom style

An *Atoms* section must appear in the data file if natoms > 0 in the header section. The atoms can be listed in any order. These are the line formats for each *atom style* in LAMMPS. As discussed below, each line can optionally have 3 flags (nx,ny,nz) appended to it, which indicate which image of a periodic simulation box the atom is in. These may be important to include for some kinds of analysis.

Note

For orthogonal and restricted and general triclinic simulation boxes, the atom coordinates (x,y,z) listed in this section should be inside the corresponding simulation box. For restricted triclinic boxes that means the parallelepiped defined by the *xlo xhi*, *ylo yhi*, *zlo zhi*, and *xy xz yz*, keywords. For general triclinic boxes that means the parallelepiped defined by the 3 edge vectors and origin specified by the *avec*, *bvec*, *cvec*, and *abc origin* header keywords. See the discussion in the header section above about how atom coordinates outside the simulation box are (or are not) remapped to be inside the box.

angle	atom-ID molecule-ID atom-type x y z
atomic	atom-ID atom-type x y z
body	atom-ID atom-type bodyflag mass x y z
bond	atom-ID molecule-ID atom-type x y z
bpm/sphere	atom-ID molecule-ID atom-type diameter density x y z
charge	atom-ID atom-type q x y z
dielectric	atom-ID atom-type q x y z mux muy muz area ed em epsilon curvature
dipole	atom-ID atom-type q x y z mux muy muz
dpd	atom-ID atom-type theta x y z
edpd	atom-ID atom-type edpd_temp edpd_cv x y z
electron	atom-ID atom-type q espin eradius x y z
ellipsoid	atom-ID atom-type ellipsoidflag density x y z
full	atom-ID molecule-ID atom-type q x y z
line	atom-ID molecule-ID atom-type lineflag density x y z
mdpd	atom-ID atom-type rho x y z
molecular	atom-ID molecule-ID atom-type x y z
peri	atom-ID atom-type volume density x y z
rheo	atom-ID atom-type status rho x y z
rheo/thermal	atom-ID atom-type status rho energy x y z
smd	atom-ID atom-type molecule volume mass kradius cradius x0 y0 z0 x y z
sph	atom-ID atom-type rho esph cv x y z
sphere	atom-ID atom-type diameter density x y z
spin	atom-ID atom-type x y z spx spy spz sp
tdpd	atom-ID atom-type x y z cc1 cc2 ... ccNspecies
template	atom-ID atom-type molecule-ID template-index template-atom x y z
tri	atom-ID molecule-ID atom-type triangleflag density x y z
wavepacket	atom-ID atom-type charge espin eradius etag cs_re cs_im x y z
hybrid	atom-ID atom-type x y z sub-style1 sub-style2 ...

The per-atom values have these meanings and units, listed alphabetically:

- atom-ID = integer ID of atom
- atom-type = type of atom (1-Ntype, or type label)
- bodyflag = 1 for body particles, 0 for point particles
- ccN = chemical concentration for tDPD particles for each species (mole/volume units)
- cradius = contact radius for SMD particles (distance units)
- cs_re,cs_im = real/imaginary parts of wave packet coefficients
- cv = heat capacity (need units) for SPH particles
- density = density of particle (mass/distance^3 or mass/distance^2 or mass/distance units, depending on dimensionality of particle)
- diameter = diameter of spherical atom (distance units)
- edpd_temp = temperature for eDPD particles (temperature units)
- edpd_cv = volumetric heat capacity for eDPD particles (energy/temperature/volume units)
- ellipsoidflag = 1 for ellipsoidal particles, 0 for point particles
- eradius = electron radius (or fixed-core radius)
- esph = energy (need units) for SPH particles

- espin = electron spin (+1/-1), 0 = nuclei, 2 = fixed-core, 3 = pseudo-cores (i.e. ECP)
- etag = integer ID of electron that each wave packet belongs to
- kradius = kernel radius for SMD particles (distance units)
- lineflag = 1 for line segment particles, 0 for point or spherical particles
- mass = mass of particle (mass units)
- molecule-ID = integer ID of molecule the atom belongs to
- mux,muy,muz = components of dipole moment of atom (dipole units) (see general triclinic note below)
- q = charge on atom (charge units)
- rho = density (need units) for SPH particles
- sp = magnitude of magnetic spin of atom (Bohr magnetons)
- spx,spy,spz = components of magnetic spin of atom (unit vector) (see general triclinic note below)
- template-atom = which atom within a template molecule the atom is
- template-index = which molecule within the molecule template the atom is part of
- theta = internal temperature of a DPD particle
- triangleflag = 1 for triangular particles, 0 for point or spherical particles
- volume = volume of Peridynamic particle (distance^3 units)
- x,y,z = coordinates of atom (distance units)
- x0,y0,z0 = original (strain-free) coordinates of atom (distance units) (see general triclinic note below)

The units for these quantities depend on the unit style; see the *units* command for details.

For 2d simulations, the atom coordinate z must be specified as 0.0. If the data file is created by another program, then z values for a 2d simulation can be within epsilon of 0.0, and LAMMPS will force them to zero.

Note

If the data file defines a general triclinic box, then the following per-atom values in the list above are per-atom vectors which imply an orientation: (mux,muy,muz) and (spx,spy,spz). This means they should be specified consistent with the general triclinic box and its orientation relative to the standard x,y,z coordinate axes. For example a dipole moment 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. Likewise the (x0,y0,z0) per-atom strain-free coordinates should be inside the general triclinic simulation box as explained in the note above. See the *Howto triclinic* doc page for more details.

The atom-ID is used to identify the atom throughout the simulation and in dump files. Normally, it is a unique value from 1 to Natoms for each atom. Unique values larger than Natoms can be used, but they will cause extra memory to be allocated on each processor, if an atom map array is used, but not if an atom map hash is used; see the *atom_modify* command for details. If an atom map is not used (e.g. an atomic system with no bonds), and you don't care if unique atom IDs appear in dump files, then the atom-IDs can all be set to 0.

The atom-type can be a numeric value or an alphanumeric label. The latter is only allowed if the type label has been defined by the *labelmap* command or an Atom Type Labels section earlier in the data file. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

The molecule ID is a second identifier attached to an atom. Normally, it is a number from 1 to N, identifying which molecule the atom belongs to. It can be 0 if it is a non-bonded atom or if you don't care to keep track of molecule assignments.

The diameter specifies the size of a finite-size spherical particle. It can be set to 0.0, which means that atom is a point particle.

The ellipsoidflag, lineflag, triangleflag, and bodyflag determine whether the particle is a finite-size ellipsoid or line or triangle or body of finite size, or whether the particle is a point particle. Additional attributes must be defined for each ellipsoid, line, triangle, or body in the corresponding *Ellipsoids*, *Lines*, *Triangles*, or *Bodies* section.

The *template-index* and *template-atom* are only defined used by *atom_style template*. In this case the *molecule* command is used to define a molecule template which contains one or more molecules (as separate files). If an atom belongs to one of those molecules, its *template-index* and *template-atom* are both set to positive integers; if not the values are both 0. The *template-index* is which molecule (1 to Nmols) the atom belongs to. The *template-atom* is which atom (1 to Natoms) within the molecule the atom is.

Some pair styles and fixes and computes that operate on finite-size particles allow for a mixture of finite-size and point particles. See the doc pages of individual commands for details.

For finite-size particles, the density is used in conjunction with the particle volume to set the mass of each particle as mass = density * volume. In this context, volume can be a 3d quantity (for spheres or ellipsoids), a 2d quantity (for triangles), or a 1d quantity (for line segments). If the volume is 0.0, meaning a point particle, then the density value is used as the mass. One exception is for the body atom style, in which case the mass of each particle (body or point particle) is specified explicitly. This is because the volume of the body is unknown.

Note that for 2d simulations of spheres, this command will treat them as spheres when converting density to mass. However, they can also be modeled as 2d discs (circles) if the *set density/disc* command is used to reset their mass after the read_data command is used. A *disc* keyword can also be used with time integration fixes, such as *fix nve/sphere* and *fix nvt/sphere* to time integrate their motion as 2d discs (not 3d spheres), by changing their moment of inertia.

For atom_style hybrid, following the 5 initial values (ID,type,x,y,z), specific values for each sub-style must be listed. The order of the sub-styles is the same as they were listed in the *atom_style* command. The specific values for each sub-style are those that are not the 5 standard ones (ID,type,x,y,z). For example, for the "charge" sub-style, a "q" value would appear. For the "full" sub-style, a "molecule-ID" and "q" would appear. These are listed in the same order they appear as listed above. Thus if

```
atom_style hybrid charge sphere
```

were used in the input script, each atom line would have these fields:

```
atom-ID atom-type x y z q diameter density
```

Note that if a non-standard value is defined by multiple sub-styles, it only appears once in the atom line. E.g. the atom line for atom_style hybrid dipole full would list "q" only once, with the dipole sub-style fields; "q" does not appear with the full sub-style fields.

```
atom-ID atom-type x y z q mux muy myz molecule-ID
```

Atom lines specify the (x,y,z) coordinates of atoms. These can be inside or outside the simulation box. When the data file is read, LAMMPS wraps coordinates outside the box back into the box for dimensions that are periodic. As discussed above, if an atom is outside the box in a non-periodic dimension, it will be lost.

LAMMPS always stores atom coordinates as values which are inside the simulation box. It also stores 3 flags which indicate which image of the simulation box (in each dimension) the atom would be in if its coordinates were unwrapped across periodic boundaries. An image flag of 0 means the atom is still inside the box when unwrapped. A value of 2 means add 2 box lengths to get the unwrapped coordinate. A value of -1 means subtract 1 box length to get the unwrapped coordinate. LAMMPS updates these flags as atoms cross periodic boundaries during the simulation. The *dump* command can output atom coordinates in wrapped or unwrapped form, as well as the 3 image flags.

In the data file, atom lines (all lines or none of them) can optionally list 3 trailing integer values (nx,ny,nz), which are used to initialize the atom's image flags. If nx,ny,nz values are not listed in the data file, LAMMPS initializes them to 0. Note that the image flags are immediately updated if an atom's coordinates need to wrapped back into the simulation box.

It is only important to set image flags correctly in a data file if a simulation model relies on unwrapped coordinates for some calculation; otherwise they can be left unspecified. Examples of LAMMPS commands that use unwrapped coordinates internally are as follows:

- Atoms in a rigid body (see *fix rigid*, *fix rigid/small*) must have consistent image flags, so that when the atoms are unwrapped, they are near each other, i.e. as a single body.
- If the *replicate* command is used to generate a larger system, image flags must be consistent for bonded atoms when the bond crosses a periodic boundary. I.e. the values of the image flags should be different by 1 (in the appropriate dimension) for the two atoms in such a bond.
- If you plan to *dump* image flags and perform post-analysis that will unwrap atom coordinates, it may be important that a continued run (restarted from a data file) begins with image flags that are consistent with the previous run.

Note

If your system is an infinite periodic crystal with bonds then it is impossible to have fully consistent image flags. This is because some bonds will cross periodic boundaries and connect two atoms with the same image flag.

Atom velocities and other atom quantities not defined above are set to 0.0 when the *Atoms* section is read. Velocities can be set later by a *Velocities* section in the data file or by a *velocity* or *set* command in the input script.

Bodies section:

- one or more lines per body
- first line syntax: atom-ID Ninteger Ndouble

```
Ninteger = # of integer quantities for this particle
Ndouble = # of floating-point quantities for this particle
```

- 0 or more integer lines with total of Ninteger values
- 0 or more double lines with total of Ndouble values
- example:

```
12 3 6
2 3 2
1.0 2.0 3.0 1.0 2.0 4.0
```

- example:

```
12 0 14
1.0 2.0 3.0 1.0 2.0 4.0 1.0
2.0 3.0 1.0 2.0 4.0 4.0 2.0
```

The *Bodies* section must appear if *atom_style body* is used and any atoms listed in the *Atoms* section have a bodyflag = 1. The number of bodies should be specified in the header section via the "bodies" keyword.

Each body can have a variable number of integer and/or floating-point values. The number and meaning of the values is defined by the body style, as described in the *Howto body* doc page. The body style is given as an argument to the *atom_style body* command.

The Ninteger and Ndouble values determine how many integer and floating-point values are specified for this particle. Ninteger and Ndouble can be as large as needed and can be different for every body. Integer values are then listed next on subsequent lines. Lines are read one at a time until Ninteger values are read. Floating-point values follow on subsequent lines, Again lines are read one at a time until Ndouble values are read. Note that if there are no values of a particular type, no lines appear for that type.

The *Bodies* section must appear after the *Atoms* section.

Bond Coeffs section:

- one line per bond type
- line syntax: ID coeffs

```
ID = bond type (1-N)
coeffs = list of coeffs
```

- example:

```
4 250 1.49
```

The number and meaning of the coefficients are specific to the defined bond style. See the *bond_style* and *bond_coeff* commands for details. Coefficients can also be set via the *bond_coeff* command in the input script.

Bond Type Labels section:

- one line per bond type
- line syntax: ID label

```
ID = bond type (1-N)
label = alphanumeric type label
```

Define alphanumeric type labels for each numeric bond type. These can be used in the Bonds section in place of a numeric type, but only if the this section appears before the Angles section.

See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

BondAngle Coeffs section:

- one line per angle type
- line syntax: ID coeffs

  ID = angle type (1-N)
  coeffs = list of coeffs (see class 2 section of angle_coeff)

BondBond Coeffs section:

- one line per angle type

- line syntax: ID coeffs

  ID = angle type (1-N)
  coeffs = list of coeffs (see class 2 section of angle_coeff)

BondBond13 Coeffs section:

- one line per dihedral type
- line syntax: ID coeffs

  ID = dihedral type (1-N)
  coeffs = list of coeffs (see class 2 section of dihedral_coeff)

Bonds section:

- one line per bond
- line syntax: ID type atom1 atom2

  ID = bond number (1-Nbonds)
  type = bond type (1-Nbondtype, or type label)
  atom1,atom2 = IDs of 1st,2nd atom in bond

- example:

  ```
  12 3 17 29
  ```

The *Bonds* section must appear after the *Atoms* section.

All values in this section must be integers (1, not 1.0). However, the type can be a numeric value or an alphanumeric label. The latter is only allowed if the type label has been defined by the *labelmap* command or a Bond Type Labels section earlier in the data file. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

Dihedral Coeffs section:

- one line per dihedral type
- line syntax: ID coeffs

  ID = dihedral type (1-N)
  coeffs = list of coeffs

- example:

  ```
  3 0.6 1 0 1
  ```

The number and meaning of the coefficients are specific to the defined dihedral style. See the *dihedral_style* and *dihedral_coeff* commands for details. Coefficients can also be set via the *dihedral_coeff* command in the input script.

Dihedral Type Labels section:

- one line per dihedral type

- line syntax: ID label

```
ID = dihedral type (1-N)
label = alphanumeric type label
```

Define alphanumeric type labels for each numeric dihedral type. These can be used in the Dihedrals section in place of a numeric type, but only if the this section appears before the Dihedrals section.

See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

Dihedrals section:

- one line per dihedral
- line syntax: ID type atom1 atom2 atom3 atom4

```
ID = number of dihedral (1-Ndihedrals)
type = dihedral type (1-Ndihedraltype, or type label)
atom1,atom2,atom3,atom4 = IDs of 1st,2nd,3rd,4th atom in dihedral
```

- example:

```
12 4 17 29 30 21
```

The 4 atoms are ordered linearly within the dihedral. The *Dihedrals* section must appear after the *Atoms* section.

All values in this section must be integers (1, not 1.0). However, the type can be a numeric value or an alphanumeric label. The latter is only allowed if the type label has been defined by the *labelmap* command or a Dihedral Type Labels section earlier in the data file. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

Ellipsoids section:

- one line per ellipsoid
- line syntax: atom-ID shapex shapey shapez quatw quati quatj quatk

```
atom-ID = ID of atom which is an ellipsoid
shapex,shapey,shapez = 3 diameters of ellipsoid (distance units)
quatw,quati,quatj,quatk = quaternion components for orientation of atom
```

- example:

```
12 1 2 1 1 0 0 0
```

The *Ellipsoids* section must appear if *atom_style ellipsoid* is used and any atoms are listed in the *Atoms* section with an ellipsoidflag = 1. The number of ellipsoids should be specified in the header section via the "ellipsoids" keyword.

The 3 shape values specify the 3 diameters or aspect ratios of a finite-size ellipsoidal particle, when it is oriented along the 3 coordinate axes. They must all be non-zero values.

The values *quatw*, *quati*, *quatj*, and *quatk* set the orientation of the atom as a quaternion (4-vector). Note that the shape attributes specify the aspect ratios of an ellipsoidal particle, which is oriented by default with its x-axis along the simulation box's x-axis, and similarly for y and z. If this body is rotated (via the right-hand rule) by an angle theta around a unit vector (a,b,c), then the quaternion that represents its new orientation is given by (cos(theta/2),

a*sin(theta/2), b*sin(theta/2), c*sin(theta/2)). These 4 components are quatw, quati, quatj, and quatk as specified above. LAMMPS normalizes each atom's quaternion in case (a,b,c) is not specified as a unit vector.

If the data file defines a general triclinic box, then the quaternion for each ellipsoid should be specified for its orientation relative to the standard x,y,z coordinate axes. When the system is converted to a restricted triclinic box, the ellipsoid quaternions will be altered to reflect the new orientation of the ellipsoid.

The *Ellipsoids* section must appear after the *Atoms* section.

EndBondTorsion Coeffs section:

- one line per dihedral type
- line syntax: ID coeffs

  ID = dihedral type (1-N)
  coeffs = list of coeffs (see class 2 section of dihedral_coeff)

Improper Coeffs section:

- one line per improper type
- line syntax: ID coeffs

```
ID = improper type (1-N)
coeffs = list of coeffs
```

- example:

```
2 20 0.0548311
```

The number and meaning of the coefficients are specific to the defined improper style. See the *improper_style* and *improper_coeff* commands for details. Coefficients can also be set via the *improper_coeff* command in the input script.

Improper Type Labels section:

- one line per improper type
- line syntax: ID label

```
ID = improper type (1-N)
label = alphanumeric type label
```

Define alphanumeric type labels for each numeric improper type. These can be used in the Impropers section in place of a numeric type, but only if the this section appears before the Impropers section.

See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

Impropers section:

- one line per improper
- line syntax: ID type atom1 atom2 atom3 atom4

```
ID = number of improper (1-Nimpropers)
type = improper type (1-Nimpropertype, or type label)
atom1,atom2,atom3,atom4 = IDs of 1st,2nd,3rd,4th atom in improper
```

- example:

```
12 3 17 29 13 100
```

The ordering of the 4 atoms determines the definition of the improper angle used in the formula for each *improper style*. See the doc pages for individual styles for details.

The *Impropers* section must appear after the *Atoms* section.

All values in this section must be integers (1, not 1.0). However, the type can be a numeric value or an alphanumeric label. The latter is only allowed if the type label has been defined by the *labelmap* command or a Improper Type Labels section earlier in the data file. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

Lines section:

- one line per line segment
- line syntax: atom-ID x1 y1 x2 y2

```
atom-ID = ID of atom which is a line segment
x1,y1 = 1st end point
x2,y2 = 2nd end point
```

- example:

```
12 1.0 0.0 2.0 0.0
```

The *Lines* section must appear if *atom_style line* is used and any atoms are listed in the *Atoms* section with a lineflag = 1. The number of lines should be specified in the header section via the "lines" keyword.

The 2 end points are the end points of the line segment. They should be values close to the center point of the line segment specified in the Atoms section of the data file, even if individual end points are outside the simulation box.

The ordering of the 2 points should be such that using a right-hand rule to cross the line segment with a unit vector in the +z direction, gives an "outward" normal vector perpendicular to the line segment. I.e. normal = (c2-c1) x (0,0,1). This orientation may be important for defining some interactions.

If the data file defines a general triclinic box, then the x1,y1 and x2,y2 values for each line segment should be specified for its orientation relative to the standard x,y,z coordinate axes. When the system is converted to a restricted triclinic box, the x1,y1,x2,y2 values will be altered to reflect the new orientation of the line segment.

The *Lines* section must appear after the *Atoms* section.

Masses section:

- one line per atom type
- line syntax: ID mass

```
ID = atom type (1-N or atom type label)
mass = mass value
```

- example:

```
3 1.01
```

This defines the mass of each atom type. This can also be set via the *mass* command in the input script. This section cannot be used for atom styles that define a mass for individual atoms - e.g. *atom_style sphere*.

Using type labels instead of atom type numbers is only allowed if the type label has been defined by the *labelmap* command or a Atom Type Labels section earlier in the data file. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

MiddleBondTorsion Coeffs section:

- one line per dihedral type
- line syntax: ID coeffs

  ID = dihedral type (1-N)
  coeffs = list of coeffs (see class 2 section of dihedral_coeff)

Pair Coeffs section:

- one line per atom type
- line syntax: ID coeffs

```
ID = atom type (1-N)
coeffs = list of coeffs
```

- example:

```
3 0.022 2.35197 0.022 2.35197
```

The number and meaning of the coefficients are specific to the defined pair style. See the *pair_style* and *pair_coeff* commands for details. Since pair coefficients for types I != J are not specified, these will be generated automatically by the pair style's mixing rule. See the individual pair_style doc pages and the *pair_modify mix* command for details. Pair coefficients can also be set via the *pair_coeff* command in the input script.

PairIJ Coeffs section:

- one line per pair of atom types for all I,J with I <= J
- line syntax: ID1 ID2 coeffs

```
ID1 = atom type I = 1-N
ID2 = atom type J = I-N, with I <= J
coeffs = list of coeffs
```

- examples:

```
3 3 0.022 2.35197 0.022 2.35197
3 5 0.022 2.35197 0.022 2.35197
```

This section must have N*(N+1)/2 lines where N = # of atom types. The number and meaning of the coefficients are specific to the defined pair style. See the *pair_style* and *pair_coeff* commands for details. Since pair coefficients for types I != J are all specified, these values will turn off the default mixing rule defined by the pair style. See the individual pair_style doc pages and the *pair_modify mix* command for details. Pair coefficients can also be set via the *pair_coeff* command in the input script.

Triangles section:

- one line per triangle
- line syntax: atom-ID x1 y1 z1 x2 y2 z2 x3 y3 z3

```
atom-ID = ID of atom which is a line segment
x1,y1,z1 = 1st corner point
x2,y2,z2 = 2nd corner point
x3,y3,z3 = 3rd corner point
```

- example:

```
12 0.0 0.0 0.0 2.0 0.0 1.0 0.0 2.0 1.0
```

The *Triangles* section must appear if *atom_style tri* is used and any atoms are listed in the *Atoms* section with a triangleflag = 1. The number of lines should be specified in the header section via the "triangles" keyword.

The 3 corner points are the corner points of the triangle. They should be values close to the center point of the triangle specified in the Atoms section of the data file, even if individual corner points are outside the simulation box.

The ordering of the 3 points should be such that using a right-hand rule to go from point1 to point2 to point3 gives an "outward" normal vector to the face of the triangle. I.e. normal = (c2-c1) x (c3-c1). This orientation may be important for defining some interactions.

If the data file defines a general triclinic box, then the x1,y1,z1 and x2,y2,z2 and x3,y3,z3 values for each triangle should be specified for its orientation relative to the standard x,y,z coordinate axes. When the system is converted to a restricted triclinic box, the x1,y1,z1,x2,y2,z2,x3,y3,z3 values will be altered to reflect the new orientation of the triangle.

The *Triangles* section must appear after the *Atoms* section.

Velocities section:

- one line per atom
- line syntax: depends on atom style

all styles except those listed	atom-ID vx vy vz
electron	atom-ID vx vy vz ervel
ellipsoid	atom-ID vx vy vz lx ly lz
sphere	atom-ID vx vy vz wx wy wz
hybrid	atom-ID vx vy vz sub-style1 sub-style2 ...

where the keywords have these meanings:

```
vx,vy,vz = translational velocity of atom
lx,ly,lz = angular momentum of aspherical atom
wx,wy,wz = angular velocity of spherical atom
ervel = electron radial velocity (0 for fixed-core)
```

The velocity lines can appear in any order. This section can only be used after an *Atoms* section. This is because the *Atoms* section must have assigned a unique atom ID to each atom so that velocities can be assigned to them.

Vx, vy, vz, and ervel are in *units* of velocity. Lx, ly, lz are in units of angular momentum (distance-velocity-mass). Wx, Wy, Wz are in units of angular velocity (radians/time).

If the data file defines a general triclinic box, then each of the 3 vectors (translational velocity, angular momentum, angular velocity) should be specified for the rotated coordinate axes of the general triclinic box. See the *Howto triclinic* doc page for more details.

For atom_style hybrid, following the 4 initial values (ID,vx,vy,vz), specific values for each sub-style must be listed. The order of the sub-styles is the same as they were listed in the *atom_style* command. The sub-style specific values are those that are not the 5 standard ones (ID,vx,vy,vz). For example, for the "sphere" sub-style, "wx", "wy", "wz" values would appear. These are listed in the same order they appear as listed above. Thus if

```
atom_style hybrid electron sphere
```

were used in the input script, each velocity line would have these fields:

```
atom-ID vx vy vz ervel wx wy wz
```

Translational velocities can also be (re)set by the *velocity* command in the input script.

1.85.10 Restrictions

To read gzipped data files, you must compile LAMMPS with the -DLAMMPS_GZIP option. See the *Build settings* doc page for details.

Label maps are currently not supported when using the KOKKOS package.

1.85.11 Related commands

read_dump, *read_restart*, *create_atoms*, *write_data*, *labelmap*

1.85.12 Default

The default for all the *extra* keywords is 0.

1.86 read_dump command

1.86.1 Syntax

```
read_dump file Nstep field1 field2 ... keyword values ...
```

- file = name of dump file to read
- Nstep = snapshot timestep to read from file
- one or more fields may be appended

field = x or y or z or vx or vy or vz or q or ix or iy or iz or fx or fy or fz
  x,y,z = atom coordinates
  vx,vy,vz = velocity components
  q = charge
  ix,iy,iz = image flags in each dimension
  fx,fy,fz = force components

- zero or more keyword/value pairs may be appended
- keyword = *nfile* or *box* or *timestep* or *replace* or *purge* or *trim* or *add* or *label* or *scaled* or *wrapped* or *format*

nfile value = Nfiles = how many parallel dump files exist
box value = yes or no = replace simulation box with dump box
timestep value = yes or no = reset simulation timestep with dump timestep
replace value = yes or no = overwrite atoms with dump atoms
purge value = yes or no = delete all atoms before adding dump atoms
trim value = yes or no = trim atoms not in dump snapshot
add value = yes or keep or no = add new dump atoms to system
label value = field column
  field = one of the listed fields or id or type
  column = label on corresponding column in dump file
scaled value = yes or no = coords in dump file are scaled/unscaled
wrapped value = yes or no = coords in dump file are wrapped/unwrapped
format values = format of dump file, must be last keyword if used
  native = native LAMMPS dump file
  xyz = XYZ file
  adios [timeout value] = dump file written by the dump adios command
    timeout = specify waiting time for the arrival of the timestep when running concurrently.
              The value is a float number and is interpreted in seconds.
  molfile style path = VMD molfile plugin interface
    style = dcd or xyz or others supported by molfile plugins
    path = optional path for location of molfile plugins

1.86.2 Examples

```
read_dump dump.file 5000 x y z
read_dump dump.xyz 5 x y z box no format xyz
read_dump dump.xyz 10 x y z box no format molfile xyz "../plugins"
read_dump dump.dcd 0 x y z box yes format molfile dcd
read_dump dump.file 1000 x y z vx vy vz box yes format molfile lammpstrj /usr/local/lib/vmd/plugins/
↪LINUXAMD64/plugins/molfile
read_dump dump.file 5000 x y vx vy trim yes
read_dump dump.file 5000 x y vx vy add yes box no timestep no
read_dump ../run7/dump.file.gz 10000 x y z box yes
read_dump dump.xyz 10 x y z box no format molfile xyz ../plugins
read_dump dump.dcd 0 x y z format molfile dcd
read_dump dump.file 1000 x y z vx vy vz format molfile lammpstrj /usr/local/lib/vmd/plugins/
↪LINUXAMD64/plugins/molfile
read_dump dump.bp 5000 x y z vx vy vz format adios
read_dump dump.bp 5000 x y z vx vy vz format adios timeout 60.0
```

1.86.3 Description

Read atom information from a dump file to overwrite the current atom coordinates, and optionally the atom velocities and image flags, the simulation timestep, and the simulation box dimensions. This is useful for restarting a run from a particular snapshot in a dump file. See the *read_restart* and *read_data* commands for alternative methods to do this. Also see the *rerun* command for a means of reading multiple snapshots from a dump file.

Note that a simulation box must already be defined before using the read_dump command. This can be done by the *create_box*, *read_data*, or *read_restart* commands. The read_dump command can reset the simulation box dimensions, as explained below.

Also note that reading per-atom information from a dump snapshot is limited to the atom coordinates, velocities and image flags, as explained below. Other atom properties, which may be necessary to run a valid simulation, such as atom charge, or bond topology information for a molecular system, are not read from (or may not even be contained in) dump files. Thus this auxiliary information should be defined in the usual way, e.g. in a data file read in by a *read_data* command, before using the read_dump command, or by the *set* command, after the dump snapshot is read.

If the dump filename specified as *file* ends with ".gz", the dump file is read in gzipped format.

You can read dump files that were written (in parallel) to multiple files via the "%" wild-card character in the dump file name. If any specified dump file name contains a "%", they must all contain it. See the *dump* command for details. The "%" wild-card character is only supported by the *native* format for dump files, described next.

If reading parallel dump files, you must also use the *nfile* keyword to tell LAMMPS how many parallel files exist, via its specified *Nfiles* value.

The format of the dump file is selected through the *format* keyword. If specified, it must be the last keyword used, since all remaining arguments are passed on to the dump reader. The *native* format is for native LAMMPS dump files, written with a *dump atom* or *dump custom* command. The *xyz* format is for generic XYZ formatted dump files. These formats take no additional values.

The *molfile* format supports reading data through using the VMD molfile plugin interface. This dump reader format is only available, if the MOLFILE package has been installed when compiling LAMMPS.

The *molfile* format takes one or two additional values. The *style* value determines the file format to be used and can be any format that the molfile plugins support, such as DCD or XYZ. Note that DCD dump files can be written by LAMMPS via the *dump dcd* command. The *path* value specifies a list of directories which LAMMPS will search for the molfile plugins appropriate to the specified *style*. The syntax of the *path* value is like other search paths: it can contain multiple directories separated by a colon (or semicolon on windows). The *path* keyword is optional and defaults to ".", i.e. the current directory.

The *adios* format supports reading data that was written by the *dump adios* command. The entire dump is read in parallel across all the processes, dividing the atoms evenly among the processes. The number of writers that has written the dump file does not matter. Using the adios style for dump and read_dump is a convenient way to dump all atoms from *N* writers and read it back by *M* readers. If one is running two LAMMPS instances concurrently where one dumps data and the other is reading it with the rerun command, the timeout option can be specified to wait on the reader side for the arrival of the requested step.

Support for other dump format readers may be added in the future.

Global information is first read from the dump file, namely timestep and box information.

The dump file is scanned for a snapshot with a timestamp that matches the specified *Nstep*. This means the LAMMPS timestep the dump file snapshot was written on for the *native* or *adios* formats.

The list of timestamps available in an adios .bp file is stored in the variable *ntimestep*:

```
console

$ bpls dump.bp -d ntimestep
  uint64_t  ntimestep  5*scalar
    (0)    0 50 100 150 200
```

Note that the *xyz* and *molfile* formats do not store the timestep. For these formats, timesteps are numbered logically, in a sequential manner, starting from 0. Thus to access the 10th snapshot in an *xyz* or *mofile* formatted dump file, use *Nstep* = 9.

The dimensions of the simulation box for the selected snapshot are also read; see the *box* keyword discussion below. For the *native* format, an error is generated if the snapshot is for a triclinic box and the current simulation box is orthogonal or vice versa. A warning will be generated if the snapshot box boundary conditions (periodic, shrink-wrapped, etc) do not match the current simulation boundary conditions, but the boundary condition information in the snapshot is otherwise ignored. See the "boundary" command for more details. The *adios* reader does the same as the *native* format reader.

For the *xyz* format, no information about the box is available, so you must set the *box* flag to *no*. See details below.

For the *molfile* format, reading simulation box information is typically supported, but the location of the simulation box origin is lost and no explicit information about periodicity or orthogonal/triclinic box shape is available. The MOLFILE package makes a best effort to guess based on heuristics, but this may not always work perfectly.

Per-atom information from the dump file snapshot is then read from the dump file snapshot. This corresponds to the specified *fields* listed in the read_dump command. It is an error to specify a z-dimension field, namely *z*, *vz*, or *iz*, for a 2d simulation.

For dump files in *native* format, each column of per-atom data has a text label listed in the file. A matching label for each field must appear, e.g. the label "vy" for the field *vy*. For the *x*, *y*, *z* fields any of the following labels are considered a match:

x, xs, xu, xsu for field x
y, ys, yu, ysu for field y
z, zs, zu, zsu for field z

The meaning of xs (scaled), xu (unwrapped), and xsu (scaled and unwrapped) is explained on the *dump* command doc page. These labels are searched for in the list of column labels in the dump file, in order, until a match is found.

The dump file must also contain atom IDs, with a column label of "id".

If the *add* keyword is specified with a value of *yes* or *keep*, as discussed below, the dump file must contain atom types, with a column label of "type".

If a column label you want to read from the dump file is not a match to a specified field, the *label* keyword can be used to specify the specific column label from the dump file to associate with that field. An example is if a time-averaged coordinate is written to the dump file via the *fix ave/atom* command. The column will then have a label corresponding to the fix-ID rather than "x" or "xs". The *label* keyword can also be used to specify new column labels for fields *id* and *type*.

For dump files in *xyz* format, only the *x*, *y*, and *z* fields are supported. The dump file does not store atom IDs, so these are assigned consecutively to the atoms as they appear in the dump file, starting from 1. Thus you should ensure that order of atoms is consistent from snapshot to snapshot in the XYZ dump file. See the *dump_modify sort* command if the XYZ dump file was written by LAMMPS.

For dump files in *molfile* format, the *x*, *y*, *z*, *vx*, *vy*, and *vz* fields can be specified. However, not all molfile formats store velocities, or their respective plugins may not support reading of velocities. The molfile dump files do not store atom IDs, so these are assigned consecutively to the atoms as they appear in the dump file, starting from 1. Thus you should

ensure that order of atoms are consistent from snapshot to snapshot in the molfile dump file. See the *dump_modify sort* command if the dump file was written by LAMMPS.

The *adios* format supports all fields that the *native* format supports except for the *q* charge field. The list of fields stored in an adios .bp file is recorded in the attributes *columns* (array of short strings) and *columnstr* (space-separated single string).

```
console

$ bpls -la dump.bp column*
  string   columns          attr   = {"id", "type", "x", "y", "z", "vx", "vy", "vz"}
  string   columnstr        attr   = "id type x y z vx vy vz "
```

Information from the dump file snapshot is used to overwrite or replace properties of the current system. There are various options for how this is done, determined by the specified fields and optional keywords.

Changed in version 3Aug2022.

The timestep of the snapshot becomes the current timestep for the simulation unless the *timestep* keyword is specified with a *no* value (default setting is *yes*). See the *reset_timestep* command if you wish to change this to a different value after the dump snapshot is read.

If the *box* keyword is specified with a *yes* value, then the current simulation box dimensions are replaced by the dump snapshot box dimensions. If the *box* keyword is specified with a *no* value, the current simulation box is unchanged.

If the *purge* keyword is specified with a *yes* value, then all current atoms in the system are deleted before any of the operations invoked by the *replace*, *trim*, or *add* keywords take place.

If the *replace* keyword is specified with a *yes* value, then atoms with IDs that are in both the current system and the dump snapshot have their properties overwritten by field values. If the *replace* keyword is specified with a *no* value, atoms with IDs that are in both the current system and the dump snapshot are not modified.

If the *trim* keyword is specified with a *yes* value, then atoms with IDs that are in the current system but not in the dump snapshot are deleted. These atoms are unaffected if the *trim* keyword is specified with a *no* value.

If the *add* keyword is specified with a *no* value (default), then dump file atoms with IDs that are not in the current system are not added to the system. They are simply ignored.

If a *yes* value is specified, the atoms with new IDs are added to the system but their atom IDs are not preserved. Instead, after all the atoms are added, new IDs are assigned to them in the same manner as is described for the *create_atoms* command. Basically the largest existing atom ID in the system is identified, and all the added atoms are assigned IDs that consecutively follow the largest ID.

If a *keep* value is specified, the atoms with new IDs are added to the system and their atom IDs are preserved. This may lead to non-contiguous IDs for the combined system.

Note that atoms added via the *add* keyword will only have the attributes read from the dump file due to the *field* arguments. For example, if *x* or *y* or *z* or *q* is not specified as a field, a value of 0.0 is used for added atoms. Added atoms must have an atom type, so this value must appear in the dump file.

Any other attributes (e.g. charge or particle diameter for spherical particles) will be set to default values, the same as if the *create_atoms* command were used.

Atom coordinates read from the dump file are first converted into unscaled coordinates, relative to the box dimensions of the snapshot. These coordinates are then be assigned to an existing or new atom in the current simulation. The coordinates will then be remapped to the simulation box, whether it is the original box or the dump snapshot box. If periodic boundary conditions apply, this means the atom will be remapped back into the simulation box if necessary.

If shrink-wrap boundary conditions apply, the new coordinates may change the simulation box dimensions. If fixed boundary conditions apply, the atom will be lost if it is outside the simulation box.

For *native* format dump files, the 3 xyz image flags for an atom in the dump file are set to the corresponding values appearing in the dump file if the *ix*, *iy*, *iz* fields are specified. If not specified, the image flags for replaced atoms are not changed and image flags for new atoms are set to default values. If coordinates read from the dump file are in unwrapped format (e.g. *xu*) then the image flags for read-in atoms are also set to default values. The remapping procedure described in the previous paragraph will then change images flags for all atoms (old and new) if periodic boundary conditions are applied to remap an atom back into the simulation box.

Note

If you get a warning about inconsistent image flags after reading in a dump snapshot, it means one or more pairs of bonded atoms now have inconsistent image flags. As discussed on the *Errors common* page this may or may not cause problems for subsequent simulations. One way this can happen is if you read image flag fields from the dump file but do not also use the dump file box parameters.

LAMMPS knows how to compute unscaled and remapped coordinates for the snapshot column labels discussed above, e.g. *x*, *xs*, *xu*, *xsu*. If another column label is assigned to the *x* or *y* or *z* field via the *label* keyword, e.g. for coordinates output by the *fix ave/atom* command, then LAMMPS needs to know whether the coordinate information in the dump file is scaled and/or wrapped. This can be set via the *scaled* and *wrapped* keywords. Note that the value of the *scaled* and *wrapped* keywords is ignored for fields *x* or *y* or *z* if the *label* keyword is not used to assign a column label to that field.

The scaled/unscaled and wrapped/unwrapped setting must be identical for any of the *x*, *y*, *z* fields that are specified. Thus you cannot read *xs* and *yu* from the dump file. Also, if the dump file coordinates are scaled and the simulation box is triclinic, then all 3 of the *x*, *y*, *z* fields must be specified, since they are all needed to generate absolute, unscaled coordinates.

1.86.4 Restrictions

To read gzipped dump files, you must compile LAMMPS with the -DLAMMPS_GZIP option. See the *Build settings* doc page for details.

The *molfile* dump file formats are part of the MOLFILE package. They are only enabled if LAMMPS was built with that packages. See the *Build package* page for more info.

To write and read adios .bp files, you must compile LAMMPS with the *ADIOS* package.

1.86.5 Related commands

dump, *dump molfile*, *dump adios*, *read_data*, *read_restart*, *rerun*

1.86.6 Default

The option defaults are box = yes, timestep = yes, replace = yes, purge = no, trim = no, add = no, scaled = no, wrapped = yes, and format = native.

1.87 read_restart command

1.87.1 Syntax

```
read_restart file
```

- file = name of binary restart file to read in

1.87.2 Examples

```
read_restart save.10000
read_restart restart.*
```

1.87.3 Description

Read in a previously saved system configuration from a restart file. This allows continuation of a previous run. Details about what information is stored (and not stored) in a restart file is given below. Basically this operation will re-create the simulation box with all its atoms and their attributes as well as some related global settings, at the point in time it was written to the restart file by a previous simulation. The simulation box will be partitioned into a regular 3d grid of rectangular bricks, one per processor, based on the number of processors in the current simulation and the settings of the *processors* command. The partitioning can later be changed by the *balance* or *fix balance* commands.

Deprecated since version 23Jun2022.

Atom coordinates that are found to be outside the simulation box when reading the restart will be remapped back into the box and their image flags updated accordingly. This previously required specifying the *remap* option, but that is no longer required.

Restart files are saved in binary format to enable exact restarts, meaning that the trajectories of a restarted run will precisely match those produced by the original run had it continued on.

Some information about a restart file can be gathered directly from the command line when using LAMMPS with the *-restart2info* command line flag. On Unix-like operating systems (like Linux or macOS), one can also *configure the "file" command line program* to display basic information about a restart file

The binary restart file format was not designed with backward, forward, or cross-platform compatibility in mind, so the files are only expected to be read correctly by the same LAMMPS executable on the same platform. Changes to the architecture, compilation settings, or LAMMPS version can render a restart file unreadable or it may read the data incorrectly. If you want a more portable format, you can use the data file format as created by the *write_data* command. Binary restart files can also be converted into a data file from the command line by the LAMMPS executable that wrote them using the *-restart2data* command line flag.

Several things can prevent exact restarts due to round-off effects, in which case the trajectories in the 2 runs will slowly diverge. These include running on a different number of processors or changing certain settings such as those set by the *newton* or *processors* commands. LAMMPS will issue a warning in these cases.

Certain fixes will not restart exactly, though they should provide statistically similar results. These include *fix shake* and *fix langevin*.

Certain pair styles will not restart exactly, though they should provide statistically similar results. This is because the forces they compute depend on atom velocities, which are used at half-step values every timestep when forces are computed. When a run restarts, forces are initially evaluated with a full-step velocity, which is different than if the run had continued. These pair styles include *granular pair styles*, *pair dpd*, and *pair lubricate*.

If a restarted run is immediately different than the run which produced the restart file, it could be a LAMMPS bug, so consider *reporting it* if you think the behavior is a bug.

Because restart files are binary, they may not be portable to other machines. In this case, you can use the *-restart command-line switch* to convert a restart file to a data file.

Similar to how restart files are written (see the *write_restart* and *restart* commands), the restart filename can contain two wild-card characters. If a "*" appears in the filename, the directory is searched for all filenames that match the pattern where "*" is replaced with a timestep value. The file with the largest timestep value is read in. Thus, this effectively means, read the latest restart file. It's useful if you want your script to continue a run from where it left off. See the *run* command and its "upto" option for how to specify the run command so it does not need to be changed either.

If a "%" character appears in the restart filename, LAMMPS expects a set of multiple files to exist. The *restart* and *write_restart* commands explain how such sets are created. Read_restart will first read a filename where "%" is replaced by "base". This file tells LAMMPS how many processors created the set and how many files are in it. Read_restart then reads the additional files. For example, if the restart file was specified as save.% when it was written, then read_restart reads the files save.base, save.0, save.1, ... save.P-1, where P is the number of processors that created the restart file.

Note that P could be the total number of processors in the previous simulation, or some subset of those processors, if the *fileper* or *nfile* options were used when the restart file was written; see the *restart* and *write_restart* commands for details. The processors in the current LAMMPS simulation share the work of reading these files; each reads a roughly equal subset of the files. The number of processors which created the set can be different the number of processors in the current LAMMPS simulation. This can be a fast mode of input on parallel machines that support parallel I/O.

Here is the list of information included in a restart file, which means these quantities do not need to be re-specified in the input script that reads the restart file, though you can redefine many of these settings after the restart file is read.

- *units*
- *newton bond* (see discussion of newton command below)
- *atom style* and *atom_modify* settings id, map, sort
- *comm style* and *comm_modify* settings mode, cutoff, vel
- *timestep size* and *timestep number*
- simulation box size and shape and *boundary* settings
- atom *group* definitions
- per-type atom settings such as *mass*
- per-atom attributes including their group assignments and molecular topology attributes (bonds, angles, etc)
- force field styles (*pair*, *bond*, *angle*, etc)
- force field coefficients (*pair*, *bond*, *angle*, etc) in some cases (see below)
- *pair_modify* settings, except the compute option
- *special_bonds* settings

Here is a list of information not stored in a restart file, which means you must re-issue these commands in your input script, after reading the restart file.

- *newton pair* (see discussion of newton command below)

- *fix* commands (see below)
- *compute* commands (see below)
- *variable* commands
- *region* commands
- *neighbor list* criteria including *neigh_modify* settings
- *kspace_style* and *kspace_modify* settings
- info for *thermodynamic*, *dump*, or *restart* output

The *newton* command has two settings, one for pairwise interactions, the other for bonded. Both settings are stored in the restart file. For the bond setting, the value in the file will overwrite the current value (at the time the read_restart command is issued) and warn if the two values are not the same and the current value is not the default. For the pair setting, the value in the file will not overwrite the current value (so that you can override the previous run's value), but a warning is issued if the two values are not the same and the current value is not the default.

Note that some force field styles (pair, bond, angle, etc) do not store their coefficient info in restart files. Typically these are many-body or tabulated potentials which read their parameters from separate files. In these cases you will need to re-specify the *pair_coeff*, *bond_coeff*, etc commands in your restart input script. The doc pages for individual force field styles mention if this is the case. This is also true of *pair_style hybrid* (bond hybrid, angle hybrid, etc) commands; they do not store coefficient info.

As indicated in the above list, the *fixes* used for a simulation are not stored in the restart file. This means the new input script should specify all fixes it will use. However, note that some fixes store an internal "state" which is written to the restart file. This allows the fix to continue on with its calculations in a restarted simulation. To re-enable such a fix, the fix command in the new input script must be of the same style and use the same fix-ID as was used in the input script that wrote the restart file.

If a match is found, LAMMPS prints a message indicating that the fix is being re-enabled. If no match is found before the first run or minimization is performed by the new script, the "state" information for the saved fix is discarded. At the time the discard occurs, LAMMPS will also print a list of fixes for which the information is being discarded. See the doc pages for individual fixes for info on which ones can be restarted in this manner. Note that fixes which are created internally by other LAMMPS commands (computes, fixes, etc) will have style names which are all-capitalized, and IDs which are generated internally.

Likewise, the *computes* used for a simulation are not stored in the restart file. This means the new input script should specify all computes it will use. However, some computes create a fix internally to store "state" information that persists from timestep to timestep. An example is the *compute msd* command which uses a fix to store a reference coordinate for each atom, so that a displacement can be calculated at any later time. If the compute command in the new input script uses the same compute-ID and group-ID as was used in the input script that wrote the restart file, then it will create the same fix in the restarted run. This means the re-created fix will be re-enabled with the stored state information as described in the previous paragraph, so that the compute can continue its calculations in a consistent manner.

Note

There are a handful of commands which can be used before or between runs which may require a system initialization. Examples include the "balance", "displace_atoms", "delete_atoms", "set" (some options), and "velocity" (some options) commands. This is because they can migrate atoms to new processors. Thus they will also discard unused "state" information from fixes. You will know the discard has occurred because a list of discarded fixes will be printed to the screen and log file, as explained above. This means that if you wish to retain that info in a restarted run, you must re-specify the relevant fixes and computes (which create fixes) before those commands are used.

Some pair styles, like the *granular pair styles*, also use a fix to store "state" information that persists from timestep to timestep. In the case of granular potentials, it is contact information between pairs of touching particles. This info will

also be re-enabled in the restart script, assuming you re-use the same granular pair style.

LAMMPS allows bond interactions (angle, etc) to be turned off or deleted in various ways, which can affect how their info is stored in a restart file.

If bonds (angles, etc) have been turned off by the *fix shake* or *delete_bonds* command, their info will be written to a restart file as if they are turned on. This means they will need to be turned off again in a new run after the restart file is read.

Bonds that are broken (e.g. by a bond-breaking potential) are written to the restart file as broken bonds with a type of 0. Thus these bonds will still be broken when the restart file is read.

Bonds that have been broken by the *fix bond/break* command have disappeared from the system. No information about these bonds is written to the restart file.

1.87.4 Restrictions

none

1.87.5 Related commands

read_data, *read_dump*, *write_restart*, *restart*

1.87.6 Default

none

1.88 region command

1.88.1 Syntax

```
region ID style args keyword arg ...
```

- ID = user-assigned name for the region
- style = *delete* or *block* or *cone* or *cylinder* or *ellipsoid* or *plane* or *prism* or *sphere* or *union* or *intersect*

  delete = no args
  block args = xlo xhi ylo yhi zlo zhi
    xlo,xhi,ylo,yhi,zlo,zhi = bounds of block in all dimensions (distance units)
    xlo,xhi,ylo,yhi,zlo,zhi can be a variable
  cone args = dim c1 c2 radlo radhi lo hi
    dim = x or y or z = axis of cone
    c1,c2 = coords of cone axis in other 2 dimensions (distance units)
    radlo,radhi = cone radii at lo and hi end (distance units)
    lo,hi = bounds of cone in dim (distance units)
    c1,c2,radlo,radhi,lo,hi can be a variable (see below)
  cylinder args = dim c1 c2 radius lo hi
    dim = x or y or z = axis of cylinder
    c1,c2 = coords of cylinder axis in other 2 dimensions (distance units)

radius = cylinder radius (distance units)
c1,c2, and radius can be a variable (see below)
lo,hi = bounds of cylinder in dim (distance units)
ellipsoid args = x y z a b c
x,y,z = center of ellipsoid (distance units)
a,b,c = half the length of the principal axes of the ellipsoid (distance units)
x,y,z,a,b and c can be a variable (see below)
plane args = px py pz nx ny nz
px,py,pz = point on the plane (distance units)
nx,ny,nz = direction normal to plane (distance units)
prism args = xlo xhi ylo yhi zlo zhi xy xz yz
xlo,xhi,ylo,yhi,zlo,zhi = bounds of untilted prism (distance units)
xy = distance to tilt y in x direction (distance units)
xz = distance to tilt z in x direction (distance units)
yz = distance to tilt z in y direction (distance units)
sphere args = x y z radius
x,y,z = center of sphere (distance units)
radius = radius of sphere (distance units)
x,y,z, and radius can be a variable (see below)
union args = N reg-ID1 reg-ID2 ...
N = # of regions to follow, must be 2 or greater
reg-ID1,reg-ID2, ... = IDs of regions to join together
intersect args = N reg-ID1 reg-ID2 ...
N = # of regions to follow, must be 2 or greater
reg-ID1,reg-ID2, ... = IDs of regions to intersect

- zero or more keyword/arg pairs may be appended
- keyword = *side* or *units* or *move* or *rotate* or *open*

side value = in or out
in = the region is inside the specified geometry
out = the region is outside the specified geometry
units value = lattice or box
lattice = the geometry is defined in lattice units
box = the geometry is defined in simulation box units
move args = v_x v_y v_z
v_x,v_y,v_z = equal-style variables for x,y,z displacement of region over time (distance units)
rotate args = v_theta Px Py Pz Rx Ry Rz
v_theta = equal-style variable for rotaton of region over time (in radians)
Px,Py,Pz = origin for axis of rotation (distance units)
Rx,Ry,Rz = axis of rotation vector
open value = integer from 1-6 corresponding to face index (see below)

- accelerated styles (with same args) = *block/kk*

1.88.2 Examples

```
region 1 block -3.0 5.0 INF 10.0 INF INF
region 2 sphere 0.0 0.0 0.0 5 side out
region void cylinder y 2 3 5 -5.0 EDGE units box
region 1 prism 0 10 0 10 0 10 2 0 0
region outside union 4 side1 side2 side3 side4
region 2 sphere 0.0 0.0 0.0 5 side out move v_left v_up NULL
region openbox block 0 10 0 10 0 10 open 5 open 6 units box
region funnel cone z 10 10 2 5 0 10 open 1 units box
```

1.88.3 Description

This command defines a geometric region of space. Various other commands use regions. For example, the region can be filled with atoms via the *create_atoms* command. Or a bounding box around the region, can be used to define the simulation box via the *create_box* command. Or the atoms in the region can be identified as a group via the *group* command, or deleted via the *delete_atoms* command. Or the surface of the region can be used as a boundary wall via the *fix wall/region* command.

Commands which use regions typically test whether an atom's position is contained in the region or not. For this purpose, coordinates exactly on the region boundary are considered to be interior to the region. This means, for example, for a spherical region, an atom on the sphere surface would be part of the region if the sphere were defined with the *side in* keyword, but would not be part of the region if it were defined using the *side out* keyword. See more details on the *side* keyword below.

Normally, regions in LAMMPS are "static", meaning their geometric extent does not change with time. If the *move* or *rotate* keyword is used, as described below, the region becomes "dynamic", meaning it's location or orientation changes with time. This may be useful, for example, when thermostatting a region, via the compute temp/region command, or when the fix wall/region command uses a region surface as a bounding wall on particle motion, i.e. a rotating container.

The *delete* style removes the named region. Since there is little overhead to defining extra regions, there is normally no need to do this, unless you are defining and discarding large numbers of regions in your input script.

The lo/hi values for *block* or *cone* or *cylinder* or *prism* styles can be specified as EDGE or INF. EDGE means they extend all the way to the global simulation box boundary. Note that this is the current box boundary; if the box changes size during a simulation, the region does not. INF means a large negative or positive number (1.0e20), so it should encompass the simulation box even if it changes size. If a region is defined before the simulation box has been created (via *create_box* or *read_data* or *read_restart* commands), then an EDGE or INF parameter cannot be used. For a *prism* region, a non-zero tilt factor in any pair of dimensions cannot be used if both the lo/hi values in either of those dimensions are INF. E.g. if the xy tilt is non-zero, then xlo and xhi cannot both be INF, nor can ylo and yhi.

Note

Regions in LAMMPS do not get wrapped across periodic boundaries, as specified by the *boundary* command. For example, a spherical region that is defined so that it overlaps a periodic boundary is not treated as 2 half-spheres, one on either side of the simulation box.

Note

Regions in LAMMPS are always 3d geometric objects, regardless of whether the *dimension* of a simulation is 2d or 3d. Thus when using regions in a 2d simulation, you should be careful to define the region so that its intersection with the 2d x-y plane of the simulation has the 2d geometric extent you want.

For style *cone*, an axis-aligned cone is defined which is like a *cylinder* except that two different radii (one at each end) can be defined. Either of the radii (but not both) can be 0.0.

For style *cone* and *cylinder*, the c1,c2 params are coordinates in the 2 other dimensions besides the cylinder axis dimension. For dim = x, c1/c2 = y/z; for dim = y, c1/c2 = x/z; for dim = z, c1/c2 = x/y. Thus the third example above specifies a cylinder with its axis in the y-direction located at x = 2.0 and z = 3.0, with a radius of 5.0, and extending in the y-direction from -5.0 to the upper box boundary.

Added in version 4May2022.

For style *ellipsoid*, an axis-aligned ellipsoid is defined. The ellipsoid has its center at (x,y,z) and is defined by 3 axis-aligned vectors given by A = (a,0,0); B = (0,b,0); C = (0,0,c). Note that although the ellipsoid is specified as axis-aligned it can be rotated via the optional *rotate* keyword.

For style *plane*, a plane is defined which contain the point (px,py,pz) and has a normal vector (nx,ny,nz). The normal vector does not have to be of unit length. The "inside" of the plane is the half-space in the direction of the normal vector; see the discussion of the *side* option below.

For style *prism*, a parallelepiped is defined (it's too hard to spell parallelepiped in an input script!). The parallelepiped has its "origin" at (xlo,ylo,zlo) and is defined by 3 edge vectors starting from the origin given by A = (xhi-xlo,0,0); B = (xy,yhi-ylo,0); C = (xz,yz,zhi-zlo). *Xy,xz,yz* can be 0.0 or positive or negative values and are called "tilt factors" because they are the amount of displacement applied to faces of an originally orthogonal box to transform it into the parallelepiped.

A prism region that will be used with the *create_box* command to define a triclinic simulation box must have tilt factors (xy,xz,yz) that do not skew the box more than half the distance of corresponding the parallel box length. For example, if xlo = 2 and xhi = 12, then the x box length is 10 and the xy tilt factor must be between -5 and 5. Similarly, both xz and yz must be between -(xhi-xlo)/2 and +(yhi-ylo)/2. Note that this is not a limitation, since if the maximum tilt factor is 5 (as in this example), then configurations with tilt = . . . , -15, -5, 5, 15, 25, . . . are all geometrically equivalent.

For style *sphere*, a sphere is defined with its center at (x,y,z) and with radius as its radius.

The *radius* value for styles *sphere* and *cylinder*, and the parameters a,b,c for style *ellipsoid*, can each be specified as an equal-style *variable*. Likewise, for style *sphere* and *ellipsoid* the x-, y-, and z- coordinates of the center of the sphere/ellipsoid can be specified as an equal-style variable. And for style *cylinder* the two center positions c1 and c2 for the location of the cylinder axes can be specified as a equal-style variable. For style *cone* all properties can be defined via equal-style variables.

If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the radius of the region.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent radius or have a time dependent position of the sphere or cylinder region.

See the *Howto tricilinc* page for a geometric description of triclinic boxes, as defined by LAMMPS, and how to transform these parameters to and from other commonly used triclinic representations.

The *union* style creates a region consisting of the volume of all the listed regions combined. The *intersect* style creates a region consisting of the volume that is common to all the listed regions.

Note

The *union* and *intersect* regions operate by invoking methods from their list of sub-regions. Thus you cannot delete the sub-regions after defining a *union* or *intersection* region.

The *side* keyword determines whether the region is considered to be inside or outside of the specified geometry. Using this keyword in conjunction with *union* and *intersect* regions, complex geometries can be built up. For example, if

the interior of two spheres were each defined as regions, and a *union* style with *side* = out was constructed listing the region-IDs of the 2 spheres, the resulting region would be all the volume in the simulation box that was outside both of the spheres.

The *units* keyword determines the meaning of the distance units used to define the region for any argument above listed as having distance units. It also affects the scaling of the velocity vector specified with the *vel* keyword, the amplitude vector specified with the *wiggle* keyword, and the rotation point specified with the *rotate* keyword, since they each involve a distance metric.

A *box* value selects standard distance units as defined by the *units* command, e.g. Angstroms for units = real or metal. A *lattice* value means the distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacings which are used as follows:

- For style *block*, the lattice spacing in dimension x is applied to xlo and xhi, similarly the spacings in dimensions y,z are applied to ylo/yhi and zlo/zhi.
- For style *cone*, the lattice spacing in argument *dim* is applied to lo and hi. The spacings in the two radial dimensions are applied to c1 and c2. The two cone radii are scaled by the lattice spacing in the dimension corresponding to c1.
- For style *cylinder*, the lattice spacing in argument *dim* is applied to lo and hi. The spacings in the two radial dimensions are applied to c1 and c2. The cylinder radius is scaled by the lattice spacing in the dimension corresponding to c1.
- For style *ellipsoid*, the lattice spacing in dimensions x,y,z are applied to the ellipsoid center x,y,z. The spacing in dimensions x,y,z are applied to the ellipsoid radii a,b,c respectively.
- For style *plane*, the lattice spacing in dimension x is applied to px and nx, similarly the spacings in dimensions y,z are applied to py/ny and pz/nz.
- For style *prism*, the lattice spacing in dimension x is applied to xlo and xhi, similarly for ylo/yhi and zlo/zhi. The lattice spacing in dimension x is applied to xy and xz, and the spacing in dimension y to yz.
- For style *sphere*, the lattice spacing in dimensions x,y,z are applied to the sphere center x,y,z. The spacing in dimension x is applied to the sphere radius.

If the *move* or *rotate* keywords are used, the region is "dynamic", meaning its location or orientation changes with time. These keywords cannot be used with a *union* or *intersect* style region. Instead, the keywords should be used to make the individual sub-regions of the *union* or *intersect* region dynamic. Normally, each sub-region should be "dynamic" in the same manner (e.g. rotate around the same point), though this is not a requirement.

The *move* keyword allows one or more *equal-style variables* to be used to specify the x,y,z displacement of the region, typically as a function of time. A variable is specified as v_name, where name is the variable name. Any of the three variables can be specified as NULL, in which case no displacement is calculated in that dimension.

Note that equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a region displacement that change as a function of time or spans consecutive runs in a continuous fashion. For the latter, see the *start* and *stop* keywords of the *run* command and the *elaplong* keyword of *thermo_style custom* for details.

For example, these commands would displace a region from its initial position, in the positive x direction, effectively at a constant velocity:

```
variable dx equal ramp(0,10)
region 2 sphere 10.0 10.0 0.0 5 move v_dx NULL NULL
```

Note that the initial displacement is 0.0, though that is not required.

Either of these variables would "wiggle" the region back and forth in the y direction:

```
variable dy equal swiggle(0,5,100)
variable dysame equal 5*sin(2*PI*elaplong*dt/100)
region 2 sphere 10.0 10.0 0.0 5 move NULL v_dy NULL
```

The *rotate* keyword rotates the region around a rotation axis *R* = (Rx,Ry,Rz) that goes through a point *P* = (Px,Py,Pz). The rotation angle is calculated, presumably as a function of time, by a variable specified as v_theta, where theta is the variable name. The variable should generate its result in radians. The direction of rotation for the region around the rotation axis is consistent with the right-hand rule: if your right-hand thumb points along *R*, then your fingers wrap around the axis in the direction of rotation.

The *move* and *rotate* keywords can be used together. In this case, the displacement specified by the *move* keyword is applied to the *P* point of the *rotate* keyword.

The *open* keyword can be used (multiple times) to indicate that one or more faces of the region are ignored for purposes of particle/wall interactions. This keyword is only relevant for regions used by the *fix wall/region* and *fix wall/gran/region* commands. It can be used to create "open" containers where only some of the region faces are walls. For example, a funnel can be created with a *cone* style region that has an open face at the smaller radius for particles to flow out, or at the larger radius for pouring particles into the cone, or both.

Note that using the *open* keyword partly overrides the *side* keyword, since both exterior and interior surfaces of an open region are tested for particle contacts. The exception to this is a *union* or *intersect* region which includes an open sub-region. In that case the *side* keyword is still used to define the union/intersect region volume, and the *open* settings are only applied to the individual sub-regions that use them.

The indices specified as part of the *open* keyword have the following meanings:

For style *block*, indices 1-6 correspond to the xlo, xhi, ylo, yhi, zlo, zhi surfaces of the block. I.e. 1 is the yz plane at x = xlo, 2 is the yz-plane at x = xhi, 3 is the xz plane at y = ylo, 4 is the xz plane at y = yhi, 5 is the xy plane at z = zlo, 6 is the xy plane at z = zhi). In the second-to-last example above, the region is a box open at both xy planes.

For style *prism*, values 1-6 have the same mapping as for style *block*. I.e. in an untilted *prism*, *open* indices correspond to the xlo, xhi, ylo, yhi, zlo, zhi surfaces.

For style *cylinder*, index 1 corresponds to the flat end cap at the low coordinate along the cylinder axis, index 2 corresponds to the high-coordinate flat end cap along the cylinder axis, and index 3 is the curved cylinder surface. For example, a *cylinder* region with *open 1 open 2* keywords will be open at both ends (e.g. a section of pipe), regardless of the cylinder orientation.

For style *cone*, the mapping is the same as for style *cylinder*. Index 1 is the low-coordinate flat end cap, index 2 is the high-coordinate flat end cap, and index 3 is the curved cone surface. In the last example above, a *cone* region is defined along the z-axis that is open at the zlo value (e.g. for use as a funnel).

For all other styles, the *open* keyword is ignored. As indicated above, this includes the *intersect* and *union* regions, though their sub-regions can be defined with the *open* keyword.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

Note

Currently, only *block* style regions are supported by Kokkos. The code using the region (such as a fix or compute) must also be supported by Kokkos or no acceleration will occur.

1.88.4 Restrictions

A prism cannot be of 0.0 thickness in any dimension; use a small z thickness for 2d simulations. For 2d simulations, the xz and yz parameters must be 0.0.

1.88.5 Related commands

lattice, *create_atoms*, *delete_atoms*, *group*

1.88.6 Default

The option defaults are side = in, units = lattice, and no move or rotation.

1.89 replicate command

1.89.1 Syntax

```
replicate nx ny nz keyword ...
```

nx,ny,nz = replication factors in each dimension

- zero or more keywords may be appended
- keyword = *bbox* or *bond/periodic*

  bbox = use a bounding-box algorithm which is faster for large proc counts
  bond/periodic = use an algorithm that correctly replicates periodic bond loops

1.89.2 Examples

For examples of replicating simple linear polymer chains (periodic or non-periodic) or periodic carbon nanotubes, see examples/replicate.

```
replicate 2 3 2
replicate 2 3 2 bbox
replicate 2 3 2 bond/periodic
```

1.89.3 Description

Replicate the current system one or more times in each dimension. For example, replication factors of 2,2,2 will create a simulation with 8x as many atoms by doubling the size of the simulation box in each dimension. A replication factor of 1 leaves the simulation domain unchanged in that dimension.

When the new simulation box is created it is partitioned into a regular 3d grid of rectangular bricks, one per processor, based on the number of processors being used and the settings of the *processors* command. The partitioning can be changed by subsequent *balance* or *fix balance* commands.

All properties of each atom are replicated (except per-atom fix data, see the Restrictions section below). This includes their velocities, which may or may not be desirable. New atom IDs are assigned to new atoms, as are new molecule IDs. Bonds and other topology interactions are created between pairs of new atoms as well as between old and new atoms.

Note

The bond discussion which follows only refers to models with permanent covalent bonds typically defined in LAMMPS via a data file. It is not relevant to systems modeled with many-body potentials which can define bonds on-the-fly, based on the current positions of nearby atoms, e.g. models using the *AIREBO* or *ReaxFF* potentials.

If the *bond/periodic* keyword is not specified, bond replication is done by using the image flag for each atom to "unwrap" it out of the periodic box before replicating it. After replication is performed, atoms outside the new periodic box are wrapped back into it. This assigns correct images flags to all atoms in the system. For this to work, all original atoms in the original simulation box must have consistent image flags. This means that if two atoms have a bond between them which crosses a periodic boundary, their respective image flags will differ by 1 in that dimension.

Image flag consistency is not possible if a system has a periodic bond loop, meaning there is a chain of bonds which crosses an entire dimension and re-connects to itself across a periodic boundary. In this case you MUST use the *bond/periodic* keyword to correctly replicate the system. This option zeroes the image flags for all atoms and uses a different algorithm to find new (nearby) bond neighbors in the replicated system. In the final replicated system all image flags are zero (in each dimension).

Note

LAMMPS does not check for image flag consistency before performing the replication (it does issue a warning about this before a simulation is run). If the original image flags are inconsistent, the replicated system will also have inconsistent image flags, but will otherwise be correctly replicated. This is NOT the case if there is a periodic bond loop. See the next note.

Note

LAMMPS does not check for periodic bond loops. If you use the *bond/periodic* keyword for a system without periodic bond loops, the system will be correctly replicated, but image flag information will be lost (which may or may not be important to your model). If you do not use the *bond/periodic* keyword for a system with periodic bond loops, the replicated system will have invalid bonds (typically very long), resulting in bad dynamics.

If possible, the *bbox* keyword should be used when running on a large number of processors, as it can result in a substantial speed-up for the replication operation. It uses a bounding box to only check atoms in replicas that overlap with each processor's new subdomain when assigning atoms to processors. It also preserves image flag information. The only drawback to the *bbox* option is that it requires a temporary use of more memory. Each processor must be able to store all atoms (and their per-atom data) in the original system, before it is replicated.

Note

The algorithm used by the *bond/periodic* keyword builds on the algorithm used by the *bbox* keyword and thus has the same memory requirements. If you specify only the *bond/peridoic* keyword it will internally set the *bbox* keyword as well.

1.89.4 Restrictions

A 2d simulation cannot be replicated in the z dimension.

If a simulation is non-periodic in a dimension, care should be used when replicating it in that dimension, as it may generate atoms nearly on top of each other.

If the current simulation was read in from a restart file (before a run is performed), there must not be any fix information stored in the file for individual atoms. Similarly, no fixes can be defined at the time the replicate command is used that require vectors of atom information to be stored. This is because the replicate command does not know how to replicate that information for new atoms it creates.

To work around this restriction two options are possible. (1) Fixes which use the stored data in the restart file can be defined before replication and then deleted via the *unfix* command and re-defined after it. Or (2) the restart file can be converted to a data file (which deletes the stored fix information) and fixes defined after the replicate command. In both these scenarios, the per-atom fix information in the restart file is lost.

1.89.5 Related commands

none

1.89.6 Default

No settings for using the *bbox* or *bond/periodic* algorithms.

1.90 rerun command

1.90.1 Syntax

```
rerun file1 file2 ... keyword args ...
```

- file1,file2,... = dump file(s) to read
- one or more keywords may be appended, keyword *dump* must appear and be last

```
keyword = first or last or every or skip or start or stop or post or dump
 first args = Nfirst
   Nfirst = dump timestep to start on
 last args = Nlast
   Nlast = dumptimestep to stop on
 every args = Nevery
   Nevery = read snapshots matching every this many timesteps
```

skip args = Nskip
  Nskip = read one out of every Nskip snapshots
start args = Nstart
  Nstart = timestep on which pseudo run will start
stop args = Nstop
  Nstop = timestep to which pseudo run will end
post value = yes or no
dump args = same as read_dump command starting with its field arguments

1.90.2 Examples

```
rerun dump.file dump x y z vx vy vz
rerun dump1.txt dump2.txt first 10000 every 1000 dump x y z
rerun dump.vels dump x y z vx vy vz box yes format molfile lammpstrj
rerun dump.dcd dump x y z box no format molfile dcd
rerun ../run7/dump.file.gz skip 2 dump x y z box yes
rerun dump.bp dump x y z box no format adios
rerun dump.bp dump x y z vx vy vz format adios timeout 10.0
```

1.90.3 Description

Perform a pseudo simulation run where atom information is read one snapshot at a time from a dump file(s), and energies and forces are computed on the shapshot to produce thermodynamic or other output.

This can be useful in the following kinds of scenarios, after an initial simulation produced the dump file:

- Compute the energy and forces of snapshots using a different potential.
- Calculate one or more diagnostic quantities on the snapshots that were not computed in the initial run. These can also be computed with settings not used in the initial run, e.g. computing an RDF via the *compute rdf* command with a longer cutoff than was used initially.
- Calculate the portion of per-atom forces resulting from a subset of the potential. E.g. compute only Coulombic forces. This can be done by only defining only a Coulombic pair style in the rerun script. Doing this in the original script would result in different (bad) dynamics.

Conceptually, using the rerun command is like running an input script that has a loop in it (see the *next* and *jump* commands). Each iteration of the loop reads one snapshot from the dump file via the *read_dump* command, sets the timestep to the appropriate value, and then invokes a *run* command for zero timesteps to simply compute energy and forces, and any other *thermodynamic output* or diagnostic info you have defined. This computation also invokes any fixes you have defined that apply constraints to the system, such as *fix shake* or *fix indent*.

Note that a simulation box must already be defined before using the rerun command. This can be done by the *create_box*, *read_data*, or *read_restart* commands.

Also note that reading per-atom information from dump snapshots is limited to the atom coordinates, velocities and image flags as explained in the *read_dump* command. Other atom properties, which may be necessary to compute energies and forces, such as atom charge, or bond topology information for a molecular system, are not read from (or even contained in) dump files. Thus this auxiliary information should be defined in the usual way, e.g. in a data file read in by a *read_data* command, before using the rerun command.

Also note that the frequency of thermodynamic or dump output from the rerun simulation will depend on settings made in the rerun script, the same as for output from any LAMMPS simulation. See further info below as to what that means if the timesteps for snapshots read from dump files do not match the specified output frequency.

If more than one dump file is specified, the dump files are read one after the other in the order specified. It is assumed that snapshot timesteps will be in ascending order. If a snapshot is encountered that is not in ascending order, it will skip the snapshot until it reads one that is. This allows skipping of a duplicate snapshot (same timestep), e.g. that appeared at the end of one file and beginning of the next. However if you specify a series of dump files in an incorrect order (with respect to the timesteps they contain), you may skip large numbers of snapshots.

Note that the dump files specified as part of the *dump* keyword can be parallel files, i.e. written as multiple files either per processor and/or per snapshot. If that is the case they will also be read in parallel which can make the rerun command operate dramatically faster for large systems. See the page for the *read_dump* and *dump* commands which describe how to read and write parallel dump files.

The *first*, *last*, *every*, *skip* keywords determine which snapshots are read from the dump file(s). Snapshots are skipped until they have a timestep >= *Nfirst*. When a snapshot with a timestep > *Nlast* is encountered, the rerun command finishes. Note that the defaults for *first* and *last* are to read all snapshots. If the *every* keyword is set to a value > 0, then only snapshots with timesteps that are a multiple of *Nevery* are read (the first snapshot is always read). If *Nevery* = 0, then this criterion is ignored, i.e. every snapshot is read that meets the other criteria. If the *skip* keyword is used, then after the first snapshot is read, every Nth snapshot is read, where N = *Nskip*. E.g. if *Nskip* = 3, then only 1 out of every 3 snapshots is read, assuming the snapshot timestep is also consistent with the other criteria.

Note

Not all dump formats contain the timestep and not all dump readers support reading it. In that case individual snapshots are assigned consecutive timestep numbers starting at 1.

The *start* and *stop* keywords do not affect which snapshots are read from the dump file(s). Rather, they have the same meaning that they do for the *run* command. They only need to be defined if (a) you are using a *fix* command that changes some value over time, and (b) you want the reference point for elapsed time (from start to stop) to be different than the *first* and *last* settings. See the page for individual fixes to see which ones can be used with the *start/stop* keywords. Note that if you define neither of the *start*/*stop* or *first*/*last* keywords, then LAMMPS treats the pseudo run as going from 0 to a huge value (effectively infinity). This means that any quantity that a fix scales as a fraction of elapsed time in the run, will essentially remain at its initial value. Also note that an error will occur if you read a snapshot from the dump file with a timestep value larger than the *stop* setting you have specified.

The *post* keyword can be used to minimize the output to the screen that happens after a *rerun* command, similar to the post keyword of the *run command*. It is set to *no* by default.

The *dump* keyword is required and must be the last keyword specified. Its arguments are passed internally to the *read_dump* command. The first argument following the *dump* keyword should be the *field1* argument of the *read_dump* command. See the *read_dump* page for details on the various options it allows for extracting information from the dump file snapshots, and for using that information to alter the LAMMPS simulation.

In general, a LAMMPS input script that uses a rerun command can include and perform all the usual operations of an input script that uses the *run* command. There are a few exceptions and points to consider, as discussed here.

Fixes that perform time integration, such as *fix nve* or *fix npt* are not invoked, since no time integration is performed. Fixes that perturb or constrain the forces on atoms will be invoked, just as they would during a normal run. Examples are *fix indent* and *fix langevin*. So you should think carefully as to whether that makes sense for the manner in which you are reprocessing the dump snapshots.

If you only want the rerun script to perform an analysis that does not involve pair interactions, such as use compute msd to calculated displacements over time, you do not need to define a *pair style*, which may also mean neighbor lists will not need to be calculated which saves time. The *comm_modify cutoff* command can also be used to ensure ghost atoms are acquired from far enough away for operations like bond and angle evaluations, if no pair style is being used.

Every time a snapshot is read, the timestep for the simulation is reset, as if the *reset_timestep* command were used. This command has some restrictions as to what fixes can be defined. See its documentation page for details. For example, the *fix deposit* and *fix dt/reset* fixes are in this category. They also make no sense to use with a rerun command.

If time-averaging fixes like *fix ave/time* are used, they are invoked on timesteps that are a function of their *Nevery*, *Nrepeat*, and *Nfreq* settings. As an example, see the *fix ave/time* page for details. You must ensure those settings are consistent with the snapshot timestamps that are read from the dump file(s). If an averaging fix is not invoked on a timestep it expects to be, LAMMPS will flag an error.

The various forms of LAMMPS output, as defined by the *thermo_style*, *thermo*, *dump*, and *restart* commands occur with specified frequency, e.g. every N steps. If the timestep for a dump snapshot is not a multiple of N, then it will be read and processed, but no output will be produced. If you want output for every dump snapshot, you can simply use N=1 for an output frequency, e.g. for thermodynamic output or new dump file output.

1.90.4 Restrictions

The *rerun* command is subject to all restrictions of the *read_dump* command.

1.90.5 Related commands

read_dump

1.90.6 Default

The option defaults are first = 0, last = a huge value (effectively infinity), start = same as first, stop = same as last, every = 0, skip = 1, post = no;

1.91 reset_atoms command

1.91.1 Syntax

```
reset_atoms property arguments ...
```

- property = *id* or *image* or *mol*
- additional arguments depend on the property

  ```
  reset_atoms id keyword value ...
  ```

  - zero or more keyword/value pairs can be appended
  - keyword = *sort*

    sort value = yes or no

  ```
  reset_atoms image group-ID
  ```

  - group-ID = ID of group of atoms whose image flags will be reset

```
reset_atoms mol group-ID keyword value ...
```

- group-ID = ID of group of atoms whose molecule IDs will be reset
- zero or more keyword/value pairs can be appended
- keyword = *compress* or *offset* or *single*

  compress value = yes or no
  offset value = Noffset >= -1
  single value = yes or no to treat single atoms (no bonds) as molecules

1.91.2 Examples

```
reset_atoms id
reset_atoms id sort yes
reset_atoms image all
reset_atoms image mobile
reset_atoms mol all
reset_atoms mol all offset 10 single yes
reset_atoms mol solvent compress yes offset 100
reset_atoms mol solvent compress no
```

1.91.3 Description

Added in version 22Dec2022.

The *reset_atoms* command resets the values of a specified atom property. In contrast to the *set* command, it does this in a collective manner which resets the values for many atoms in a self-consistent way. This command is often useful when the simulated system has undergone significant modifications like adding or removing atoms or molecules, joining data files, changing bonds, or large-scale diffusion.

The new values can be thought of as a *reset*, similar to values atoms would have if a new data file were being read or a new simulation performed. Note that the *set* command also resets atom properties to new values, but it treats each atom independently.

The *property* setting can be *id* or *image* or *mol*. For *id*, the IDs of all the atoms are reset to contiguous values. For *image*, the image flags of atoms in the specified *group-ID* are reset so that at least one atom in each molecule is in the simulation box (image flag = 0). For *mol*, the molecule IDs of all atoms are reset to contiguous values.

More details on these operations and their arguments or optional keyword/value settings are given below.

Property: *id*

Reset atom IDs for the entire system, including all the global IDs stored for bond, angle, dihedral, improper topology data. This will create a set of IDs that are numbered contiguously from 1 to N for a N atoms system.

This can be useful to do after performing a "delete_atoms" command for a molecular system. The delete_atoms compress yes option will not perform this operation due to the existence of bond topology. It can also be useful to do after any simulation which has lost atoms, e.g. due to atoms moving outside a simulation box with fixed boundaries (see the "boundary command"), or due to evaporation (see the "fix evaporate" command).

If the *sort* keyword is used with a setting of *yes*, then the assignment of new atom IDs will be the same no matter how many processors LAMMPS is running on. This is done by first doing a spatial sort of all the atoms into bins and sorting

them within each bin. Because the set of bins is independent of the number of processors, this enables a consistent assignment of new IDs to each atom.

This can be useful to do after using the "create_atoms" command and/or "replicate" command. In general those commands do not guarantee assignment of the same atom ID to the same physical atom when LAMMPS is run on different numbers of processors. Enforcing consistent IDs can be useful for debugging or comparing output from two different runs.

Note that the spatial sort requires communication of atom IDs and coordinates between processors in an all-to-all manner. This is done efficiently in LAMMPS, but it is more expensive than how atom IDs are reset without sorting.

Note that whether sorting or not, the resetting of IDs is not a compression, where gaps in atom IDs are removed by decrementing atom IDs that are larger. Instead the IDs for all atoms are erased, and new IDs are assigned so that the atoms owned by an individual processor have consecutive IDs, as the *create_atoms* command explains.

Note

If this command is used before a *pair style* is defined, an error about bond topology atom IDs not being found may result. This is because the cutoff distance for ghost atom communication was not sufficient to find atoms in bonds, angles, etc that are owned by other processors. The *comm_modify cutoff* command can be used to correct this issue. Or you can define a pair style before using this command. If you do the former, you should unset the *comm_modify cutoff* after using *reset atoms id* so that subsequent communication is not inefficient.

Property: *image*

Reset the image flags of atoms so that at least one atom in each molecule has an image flag of 0. Molecular topology is respected so that if the molecule straddles a periodic simulation box boundary, the images flags of all atoms in the molecule will be consistent. This avoids inconsistent image flags that could result from resetting all image flags to zero with the *set* command.

Note

If the system has no bonds, there is no reason to use this command, since image flags for different atoms do not need to be consistent. Use the *set* command with its *image* keyword instead.

Only image flags for atoms in the specified *group-ID* are reset; all others remain unchanged. No check is made for whether the group covers complete molecule fragments and thus whether the command will result in inconsistent image flags.

Molecular fragments are identified by the algorithm used by the *compute fragment/atom* command. For each fragment the average of the largest and the smallest image flag in each direction across all atoms in the fragment is computed and subtracted from the current image flag in the same direction.

This can be a useful operation to perform after running longer equilibration runs of mobile systems where molecules would pass through the system multiple times and thus produce non-zero image flags.

Note

Same as explained for the *compute fragment/atom* command, molecules are identified using the current bond topology. This will **not** account for bonds broken by the *bond_style quartic* command, because this bond style does not perform a full update of the bond topology data structures within LAMMPS. In that case, using the *delete_bonds all bond 0 remove* will permanently delete such broken bonds and should thus be used first.

Property: *mol*

Reset molecule IDs for a specified group of atoms based on current bond connectivity. This will typically create a new set of molecule IDs for atoms in the group. Only molecule IDs for atoms in the specified *group-ID* are reset; molecule IDs for atoms not in the group are not changed.

For purposes of this operation, molecules are identified by the current bond connectivity in the system, which may or may not be consistent with the current molecule IDs. A molecule in this context is a set of atoms connected to each other with explicit bonds. The specific algorithm used is the one of *compute fragment/atom*. Once the molecules are identified and a new molecule ID computed for each, this command will update the current molecule ID for all atoms in the group with the new molecule ID. Note that if the group excludes atoms within molecules, one (physical) molecule may become two or more (logical) molecules. For example if the group excludes atoms in the middle of a linear chain, then each end of the chain is considered an independent molecule and will be assigned a different molecule ID.

This can be a useful operation to perform after running reactive molecular dynamics run with *fix bond/react*, *fix bond/create*, or *fix bond/break*, all of which can change molecule topologies. It can also be useful after molecules have been deleted with the *delete_atoms* command or after a simulation which has lost molecules, e.g. via the *fix evaporate* command.

The *compress* keyword determines how new molecule IDs are computed. If the setting is *yes* (the default) and there are N molecules in the group, the new molecule IDs will be a set of N contiguous values. See the *offset* keyword for details on selecting the range of these values. If the setting is *no*, the molecule ID of every atom in the molecule will be set to the smallest atom ID of any atom in the molecule.

The *single* keyword determines whether single atoms (not bonded to another atom) are treated as one-atom molecules or not, based on the *yes* or *no* setting. If the setting is *no* (the default), their molecule IDs are set to 0. This setting can be important if the new molecule IDs will be used as input to other commands such as *compute chunk/atom molecule* or *fix rigid molecule*.

The *offset* keyword is only used if the *compress* setting is *yes*. Its default value is *Noffset* = -1. In that case, if the specified group is *all*, then the new compressed molecule IDs will range from 1 to N. If the specified group is not *all* and the largest molecule ID of atoms outside that group is M, then the new compressed molecule IDs will range from M+1 to M+N, to avoid collision with existing molecule IDs. If an *Noffset* >= 0 is specified, then the new compressed molecule IDs will range from *Noffset*+1 to *Noffset*+N. If the group is not *all* there may be collisions with the molecule IDs of other atoms.

Note

Same as explained for the *compute fragment/atom* command, molecules are identified using the current bond topology. This will **not** account for bonds broken by the *bond_style quartic* command, because this bond style does not perform a full update of the bond topology data structures within LAMMPS. In that case, using the *delete_bonds all bond 0 remove* will permanently delete such broken bonds and should thus be used first.

1.91.4 Restrictions

The *image* property can only be used when the atom style supports bonds.

1.91.5 Related commands

compute fragment/atom, *fix bond/react*, *fix bond/create*, *fix bond/break*, *fix evaporate*, *delete_atoms*, *delete_bonds*

1.91.6 Defaults

For property *id*, the default keyword setting is sort = no.

For property *mol*, the default keyword settings are compress = yes, single = no, and offset = -1.

1.92 reset_timestep command

1.92.1 Syntax

```
reset_timestep N keyword values ...
```

- N = timestep number
- zero or more keyword/value pairs may be appended
- keyword = *time*

  time value = atime
    atime = accumulated simulation time

1.92.2 Examples

```
reset_timestep 0
reset_timestep 4000000
reset_timestep 1000 time 100.0
```

1.92.3 Description

Set the timestep counter to the specified value. This command usually comes after the timestep has been set by reading a restart file via the *read_restart* command, or a previous simulation run or minimization advanced the timestep.

The optional *time* keyword allows to also set the accumulated simulation time. This is usually the number of timesteps times the size of the timestep, but when using variable size timesteps with *fix dt/reset* it can differ.

The *read_data* and *create_box* commands set the timestep to 0; the *read_restart* command sets the timestep to the value it had when the restart file was written. The same applies to the accumulated simulation time.

1.92.4 Restrictions

This command cannot be used when any fixes are defined that keep track of elapsed time to perform certain kinds of time-dependent operations. Examples are the *fix deposit* and *fix dt/reset* commands. The former adds atoms on specific timesteps. The latter keeps track of accumulated time.

Various fixes use the current timestep to calculate related quantities. If the timestep is reset, this may produce unexpected behavior, but LAMMPS allows the fixes to be defined even if the timestep is reset. For example, commands which thermostat the system, e.g. *fix nvt*, allow you to specify a target temperature which ramps from Tstart to Tstop which may persist over several runs. If you change the timestep, you may induce an instantaneous change in the target temperature.

Resetting the timestep clears flags for *computes* that may have calculated some quantity from a previous run. This means these quantity cannot be accessed by a variable in between runs until a new run is performed. See the *variable* command for more details.

1.92.5 Related commands

rerun, *timestep*, *fix dt/reset*

1.92.6 Default

none

1.93 restart command

1.93.1 Syntax

```
restart 0
restart N root keyword value ...
restart N file1 file2 keyword value ...
```

- N = write a restart file on timesteps which are multiples of N
- N can be a variable (see below)
- root = filename to which timestep # is appended
- file1,file2 = two full filenames, toggle between them when writing file
- zero or more keyword/value pairs may be appended
- keyword = *fileper* or *nfile*

  fileper arg = Np
    Np = write one file for every this many processors
  nfile arg = Nf
    Nf = write this many files, one from each of Nf processors

1.93.2 Examples

```
restart 0
restart 1000 poly.restart
restart 1000 restart.*.equil
restart 10000 poly.%.1 poly.%.2 nfile 10
restart v_mystep poly.restart
```

1.93.3 Description

Write out a binary restart file with the current state of the simulation on timesteps which are a multiple of N. A value of N = 0 means do not write out any restart files, which is the default. Restart files are written in one (or both) of two modes as a run proceeds. If one filename is specified, a series of filenames will be created which include the timestep in the filename. If two filenames are specified, only 2 restart files will be created, with those names. LAMMPS will toggle between the 2 names as it writes successive restart files.

Note that you can specify the restart command twice, once with a single filename and once with two filenames. This would allow you, for example, to write out archival restart files every 100000 steps using a single filename, and more frequent temporary restart files every 1000 steps, using two filenames. Using restart 0 will turn off both modes of output.

Similar to *dump* files, the restart filename(s) can contain two wild-card characters.

If a "*" appears in the single filename, it is replaced with the current timestep value. This is only recognized when a single filename is used (not when toggling back and forth). Thus, the third example above creates restart files as follows: restart.1000.equil, restart.2000.equil, etc. If a single filename is used with no "*", then the timestep value is appended. E.g. the second example above creates restart files as follows: poly.restart.1000, poly.restart.2000, etc.

If a "%" character appears in the restart filename(s), then one file is written for each processor and the "%" character is replaced with the processor ID from 0 to P-1. An additional file with the "%" replaced by "base" is also written, which contains global information. For example, the files written on step 1000 for filename restart.% would be restart.base.1000, restart.0.1000, restart.1.1000, ..., restart.P-1.1000. This creates smaller files and can be a fast mode of output and subsequent input on parallel machines that support parallel I/O. The optional *fileper* and *nfile* keywords discussed below can alter the number of files written.

Restart files are written on timesteps that are a multiple of N but not on the first timestep of a run or minimization. You can use the *write_restart* command to write a restart file before a run begins. A restart file is not written on the last timestep of a run unless it is a multiple of N. A restart file is written on the last timestep of a minimization if N > 0 and the minimization converges.

Instead of a numeric value, N can be specified as an *equal-style variable*, which should be specified as v_name, where name is the variable name. In this case, the variable is evaluated at the beginning of a run to determine the next timestep at which a restart file will be written out. On that timestep, the variable will be evaluated again to determine the next timestep, etc. Thus the variable should return timestep values. See the stagger() and logfreq() and stride() math functions for *equal-style variables*, as examples of useful functions to use in this context. Other similar math functions could easily be added as options for *equal-style variables*.

For example, the following commands will write restart files every step from 1100 to 1200, and could be useful for debugging a simulation where something goes wrong at step 1163:

```
variable    s equal stride(1100,1200,1)
restart     v_s tmp.restart
```

See the *read_restart* command for information about what is stored in a restart file.

Restart files can be read by a *read_restart* command to restart a simulation from a particular state. Because the file is binary (to enable exact restarts), it may not be readable on another machine. In this case, you can use the *-r command-line switch* to convert a restart file to a data file.

Note

Although the purpose of restart files is to enable restarting a simulation from where it left off, not all information about a simulation is stored in the file. For example, the list of fixes that were specified during the initial run is not stored, which means the new input script must specify any fixes you want to use. Even when restart information is stored in the file, as it is for some fixes, commands may need to be re-specified in the new input script, in order to re-use that information. See the *read_restart* command for information about what is stored in a restart file.

The optional *nfile* or *fileper* keywords can be used in conjunction with the "%" wildcard character in the specified restart file name(s). As explained above, the "%" character causes the restart file to be written in pieces, one piece for each of P processors. By default P = the number of processors the simulation is running on. The *nfile* or *fileper* keyword can be used to set P to a smaller value, which can be more efficient when running on a large number of processors.

The *nfile* keyword sets P to the specified Nf value. For example, if Nf = 4, and the simulation is running on 100 processors, 4 files will be written, by processors 0,25,50,75. Each will collect information from itself and the next 24 processors and write it to a restart file.

For the *fileper* keyword, the specified value of Np means write one file for every Np processors. For example, if Np = 4, every fourth processor (0,4,8,12,etc) will collect information from itself and the next 3 processors and write it to a restart file.

1.93.4 Restrictions

none

1.93.5 Related commands

write_restart, *read_restart*

1.93.6 Default

```
restart 0
```

1.94 run command

1.94.1 Syntax

```
run N keyword values ...
```

- N = # of timesteps
- zero or more keyword/value pairs may be appended

- keyword = *upto* or *start* or *stop* or *pre* or *post* or *every*

  upto value = none
  start value = N1
    N1 = timestep at which 1st run started
  stop value = N2
    N2 = timestep at which last run will end
  pre value = no or yes
  post value = no or yes
  every values = M c1 c2 ...
    M = break the run into M-timestep segments and invoke one or more commands between each segment
    c1,c2,...,cN = one or more LAMMPS commands, each enclosed in quotes
    c1 = NULL means no command will be invoked

1.94.2 Examples

```
run 10000
run 1000000 upto
run 100 start 0 stop 1000
run 1000 pre no post yes
run 100000 start 0 stop 1000000 every 1000 "print 'Protein Rg = $r'"
run 100000 every 1000 NULL
```

1.94.3 Description

Run or continue dynamics for a specified number of timesteps.

When the *run style* is *respa*, N refers to outer loop (largest) timesteps.

A value of N = 0 is acceptable; only the thermodynamics of the system are computed and printed without taking a timestep.

The *upto* keyword means to perform a run starting at the current timestep up to the specified timestep. E.g. if the current timestep is 10,000 and "run 100000 upto" is used, then an additional 90,000 timesteps will be run. This can be useful for very long runs on a machine that allocates chunks of time and terminate your job when time is exceeded. If you need to restart your script multiple times (reading in the last restart file), you can keep restarting your script with the same run command until the simulation finally completes.

The *start* or *stop* keywords can be used if multiple runs are being performed and you want a *fix* command that changes some value over time (e.g. temperature) to make the change across the entire set of runs and not just a single run. See the page for individual fixes to see which ones can be used with the *start/stop* keywords.

For example, consider this fix followed by 10 run commands:

```
fix       1 all nvt 200.0 300.0 1.0
run        1000 start 0 stop 10000
run        1000 start 0 stop 10000
...
run        1000 start 0 stop 10000
```

The NVT fix ramps the target temperature from 200.0 to 300.0 during a run. If the run commands did not have the start/stop keywords (just "run 1000"), then the temperature would ramp from 200.0 to 300.0 during the 1000 steps of each run. With the start/stop keywords, the ramping takes place over the 10000 steps of all runs together.

The *pre* and *post* keywords can be used to streamline the setup, clean-up, and associated output to the screen that happens before and after a run. This can be useful if you wish to do many short runs in succession (e.g. LAMMPS is being called as a library which is doing other computations between successive short LAMMPS runs).

By default (pre and post = yes), LAMMPS creates neighbor lists, computes forces, and imposes fix constraints before every run. And after every run it gathers and prints timings statistics. If a run is just a continuation of a previous run (i.e. no settings are changed), the initial computation is not necessary; the old neighbor list is still valid as are the forces. So if *pre* is specified as "no" then the initial setup is skipped, except for printing thermodynamic info. Note that if *pre* is set to "no" for the very first run LAMMPS performs, then it is overridden, since the initial setup computations must be done.

Note

If your input script changes the system between 2 runs, then the initial setup must be performed to ensure the change is recognized by all parts of the code that are affected. Examples are adding a *fix* or *dump* or *compute*, changing a *neighbor* list parameter, or writing restart file which can migrate atoms between processors. LAMMPS has no easy way to check if this has happened, but it is an error to use the *pre no* option in this case.

If *post* is specified as "no", the full timing summary is skipped; only a one-line summary timing is printed.

The *every* keyword provides a means of breaking a LAMMPS run into a series of shorter runs. Optionally, one or more LAMMPS commands (c1, c2, ..., cN) will be executed in between the short runs. If used, the *every* keyword must be the last keyword, since it has a variable number of arguments. Each of the trailing arguments is a single LAMMPS command, and each command should be enclosed in quotes, so that the entire command will be treated as a single argument. This will also prevent any variables in the command from being evaluated until it is executed multiple times during the run. Note that if a command itself needs one of its arguments quoted (e.g. the *print* command), then you can use a combination of single and double quotes, as in the example above or below.

The *every* keyword is a means to avoid listing a long series of runs and interleaving commands in your input script. For example, a *print* command could be invoked or a *fix* could be redefined, e.g. to reset a thermostat temperature. Or this could be useful for invoking a command you have added to LAMMPS that wraps some other code (e.g. as a library) to perform a computation periodically during a long LAMMPS run. See the *Modify* doc page for info about how to add new commands to LAMMPS. See the *Howto couple* page for ideas about how to couple LAMMPS to other codes.

With the *every* option, N total steps are simulated, in shorter runs of M steps each. After each M-length run, the specified commands are invoked. If only a single command is specified as NULL, then no command is invoked. Thus these lines:

```
variable q equal x[100]
run 6000 every 2000 "print 'Coord = $q'"
```

are the equivalent of:

```
variable q equal x[100]
run 2000
print "Coord = $q"
run 2000
print "Coord = $q"
run 2000
print "Coord = $q"
```

which does 3 runs of 2000 steps and prints the x-coordinate of a particular atom between runs. Note that the variable "$q" will be evaluated afresh each time the print command is executed.

Note that by using the line continuation character "&", the run every command can be spread across many lines, though it is still a single command:

```
run 100000 every 1000 &
  "print 'Minimum value = $a'" &
  "print 'Maximum value = $b'" &
  "print 'Temp = $c'" &
  "print 'Press = $d'"
```

If the *pre* and *post* options are set to "no" when used with the *every* keyword, then the first run will do the full setup and the last run will print the full timing summary, but these operations will be skipped for intermediate runs.

Note

You might wish to specify a command that exits the run by jumping out of the loop, e.g.

```
variable t equal temp
run 10000 every 100 "if '$t < 300.0' then 'jump SELF afterrun'"
```

However, this will not work. The run command simply executes each command one at a time each time it pauses, then continues the run.

Instead, you should use the *fix halt* command, which has additional options for how to exit the run.

1.94.4 Restrictions

When not using the *upto* keyword, the number of specified timesteps N must fit in a signed 32-bit integer, so you are limited to slightly more than 2 billion steps (2^31) in a single run. When using *upto*, N can be larger than a signed 32-bit integer, however the difference between N and the current timestep must still be no larger than 2^31 steps.

However, with or without the *upto* keyword, you can perform successive runs to run a simulation for any number of steps (ok, up to 2^63 total steps). I.e. the timestep counter within LAMMPS is a 64-bit signed integer.

1.94.5 Related commands

minimize, *run_style*, *temper*, *fix halt*

1.94.6 Default

The option defaults are start = the current timestep, stop = current timestep + N, pre = yes, and post = yes.

1.95 run_style command

1.95.1 Syntax

```
run_style style args
```

- style = *verlet* or *verlet/split* or *respa* or *respa/omp*

  verlet args = none
  verlet/split args = none
  respa args = N n1 n2 ... keyword values ...

N = # of levels of rRESPA
n1, n2, ... = loop factors between rRESPA levels (N-1 values)
zero or more keyword/value pairings may be appended to the loop factors
keyword = bond or angle or dihedral or improper or
pair or inner or middle or outer or hybrid or kspace
bond value = M
M = which level (1-N) to compute bond forces in
angle value = M
M = which level (1-N) to compute angle forces in
dihedral value = M
M = which level (1-N) to compute dihedral forces in
improper value = M
M = which level (1-N) to compute improper forces in
pair value = M
M = which level (1-N) to compute pair forces in
inner values = M cut1 cut2
M = which level (1-N) to compute pair inner forces in
cut1 = inner cutoff between pair inner and
pair middle or outer (distance units)
cut2 = outer cutoff between pair inner and
pair middle or outer (distance units)
middle values = M cut1 cut2
M = which level (1-N) to compute pair middle forces in
cut1 = inner cutoff between pair middle and pair outer (distance units)
cut2 = outer cutoff between pair middle and pair outer (distance units)
outer value = M
M = which level (1-N) to compute pair outer forces in
hybrid values = M1 M2 ... (as many values as there are hybrid sub-styles
M1 = which level (1-N) to compute the first pair_style hybrid sub-style in
M2 = which level (1-N) to compute the second pair_style hybrid sub-style in
M3,etc
kspace value = M
M = which level (1-N) to compute kspace forces in

1.95.2 Examples

```
run_style verlet
run_style respa 4 2 2 2 bond 1 dihedral 2 pair 3 kspace 4
run_style respa 4 2 2 2 bond 1 dihedral 2 inner 3 5.0 6.0 outer 4 kspace 4
run_style respa 3 4 2 bond 1 hybrid 2 2 1 kspace 3
```

1.95.3 Description

Choose the style of time integrator used for molecular dynamics simulations performed by LAMMPS.

The *verlet* style is the velocity form of the Stoermer-Verlet time integration algorithm (velocity-Verlet)

The *verlet/split* style is also a velocity-Verlet integrator, but it splits the force calculation within each timestep over 2 partitions of processors. See the *-partition command-line switch* for info on how to run LAMMPS with multiple partitions.

Specifically, this style performs all computation except the *kspace_style* portion of the force field on the first partition. This include the *pair style*, *bond style*, *neighbor list building*, *fixes* including time integration, and output. The *kspace_style* portion of the calculation is performed on the second partition.

This can lead to a significant speedup, if the number of processors can be easily increased and the fraction of time is spent in computing Kspace interactions is significant, too. The two partitions may have a different number of processors. This is most useful for the PPPM kspace_style when its performance on a large number of processors degrades due to the cost of communication in its 3d FFTs. In this scenario, splitting your P total processors into 2 subsets of processors, P1 in the first partition and P2 in the second partition, can enable your simulation to run faster. This is because the long-range forces in PPPM can be calculated at the same time as pairwise and bonded forces are being calculated *and* the parallel 3d FFTs can be faster to compute when running on fewer processors. Please note that the scenario of using fewer MPI processes to reduce communication overhead can also be implemented through using MPI with OpenMP threads via the INTEL, KOKKOS, or OPENMP package. This alternative option is typically more effective in case of a fixed number of available processors and less complex to execute.

To use the *verlet/split* style, you must define 2 partitions with the *-partition command-line switch*, where partition P1 is either the same size or an integer multiple of the size of the partition P2. Typically having P1 be 3x larger than P2 is a good choice, since the (serial) performance of LAMMPS is often best if the time spent in the Pair computation versus Kspace is a 3:1 split. The 3d processor layouts in each partition must overlay in the following sense. If P1 is a Px1 by Py1 by Pz1 grid, and P2 = Px2 by Py2 by Pz2, then Px1 must be an integer multiple of Px2, and similarly for Py1 a multiple of Py2, and Pz1 a multiple of Pz2.

Typically the best way to do this is to let the first partition choose its own optimal layout, then require the second partition's layout to match the integer multiple constraint. See the *processors* command with its *part* keyword for a way to control this, e.g.

```
processors * * * part 1 2 multiple
```

You can also use the *partition* command to explicitly specify the processor layout on each partition. E.g. for 2 partitions of 60 and 15 processors each:

```
partition yes 1 processors 3 4 5
partition yes 2 processors 3 1 5
```

When you run in 2-partition mode with the *verlet/split* style, the thermodynamic data for the entire simulation will be output to the log and screen file of the first partition, which are log.lammps.0 and screen.0 by default; see the *-plog and -pscreen command-line switches* to change this. The log and screen file for the second partition will not contain thermodynamic output beyond the first timestep of the run.

See the *Accelerator packages* page for performance details of the speed-up offered by the *verlet/split* style. One important performance consideration is the assignment of logical processors in the 2 partitions to the physical cores of a parallel machine. The *processors* command has options to support this, and strategies are discussed in *Section 5* of the manual.

The *respa* style implements the rRESPA multi-timescale integrator *(Tuckerman)* with N hierarchical levels, where level 1 is the innermost loop (shortest timestep) and level N is the outermost loop (largest timestep). The loop factor arguments specify what the looping factor is between levels. N1 specifies the number of iterations of level 1 for a single iteration of level 2, N2 is the iterations of level 2 per iteration of level 3, etc. N-1 looping parameters must be specified.

Thus with a 4-level respa setting of "2 2 2" for the 3 loop factors, you could choose to have bond interactions computed 8x per large timestep, angle interactions computed 4x, pair interactions computed 2x, and long-range interactions once per large timestep.

The *timestep* command sets the large timestep for the outermost rRESPA level. Thus if the 3 loop factors are "2 2 2" for 4-level rRESPA, and the outer timestep is set to 4.0 fs, then the inner timestep would be 8x smaller or 0.5 fs. All other LAMMPS commands that specify number of timesteps (e.g. *thermo* for thermo output every N steps, *neigh_modify delay/every* parameters, *dump* every N steps, etc) refer to the outermost timesteps.

The rRESPA keywords enable you to specify at what level of the hierarchy various forces will be computed. If not specified, the defaults are that bond forces are computed at level 1 (innermost loop), angle forces are computed where bond forces are, dihedral forces are computed where angle forces are, improper forces are computed where dihedral forces are, pair forces are computed at the outermost level, and kspace forces are computed where pair forces are. The inner, middle, outer forces have no defaults.

For fixes that support it, the rRESPA level at which a given fix is active, can be selected through the *fix_modify* command.

The *inner* and *middle* keywords take additional arguments for cutoffs that are used by the pairwise force computations. If the 2 cutoffs for *inner* are 5.0 and 6.0, this means that all pairs up to 6.0 apart are computed by the inner force. Those between 5.0 and 6.0 have their force go ramped to 0.0 so the overlap with the next regime (middle or outer) is smooth. The next regime (middle or outer) will compute forces for all pairs from 5.0 outward, with those from 5.0 to 6.0 having their value ramped in an inverse manner.

Note that you can use *inner* and *outer* without using *middle* to split the pairwise computations into two portions instead of three. Unless you are using a very long pairwise cutoff, a 2-way split is often faster than a 3-way split, since it avoids too much duplicate computation of pairwise interactions near the intermediate cutoffs.

Also note that only a few pair potentials support the use of the *inner* and *middle* and *outer* keywords. If not, only the *pair* keyword can be used with that pair style, meaning all pairwise forces are computed at the same rRESPA level. See the doc pages for individual pair styles for details.

Another option for using pair potentials with rRESPA is with the *hybrid* keyword, which requires the use of the *pair_style hybrid or hybrid/overlay* command. In this scenario, different sub-styles of the hybrid pair style are evaluated at different rRESPA levels. This can be useful, for example, to set different timesteps for hybrid coarse-grained/all-atom models. The *hybrid* keyword requires as many level assignments as there are hybrid sub-styles, which assigns each sub-style to a rRESPA level, following their order of definition in the pair_style command. Since the *hybrid* keyword operates on pair style computations, it is mutually exclusive with either the *pair* or the *inner*/*middle*/*outer* keywords.

When using rRESPA (or for any MD simulation) care must be taken to choose a timestep size(s) that ensures the Hamiltonian for the chosen ensemble is conserved. For the constant NVE ensemble, total energy must be conserved. Unfortunately, it is difficult to know *a priori* how well energy will be conserved, and a fairly long test simulation (~10 ps) is usually necessary in order to verify that no long-term drift in energy occurs with the trial set of parameters.

With that caveat, a few rules-of-thumb may be useful in selecting *respa* settings. The following applies mostly to biomolecular simulations using the CHARMM or a similar all-atom force field, but the concepts are adaptable to other problems. Without SHAKE, bonds involving hydrogen atoms exhibit high-frequency vibrations and require a timestep on the order of 0.5 fs in order to conserve energy. The relatively inexpensive force computations for the bonds, angles, impropers, and dihedrals can be computed on this innermost 0.5 fs step. The outermost timestep cannot be greater than 4.0 fs without risking energy drift. Smooth switching of forces between the levels of the rRESPA hierarchy is also necessary to avoid drift, and a 1-2 Angstrom "healing distance" (the distance between the outer and inner cutoffs) works reasonably well. We thus recommend the following settings for use of the *respa* style without SHAKE in biomolecular simulations:

```
timestep 4.0
run_style respa 4 2 2 2 inner 2 4.5 6.0 middle 3 8.0 10.0 outer 4
```

With these settings, users can expect good energy conservation and roughly a 2.5 fold speedup over the *verlet* style with a 0.5 fs timestep.

If SHAKE is used with the *respa* style, time reversibility is lost, but substantially longer time steps can be achieved. For biomolecular simulations using the CHARMM or similar all-atom force field, bonds involving hydrogen atoms exhibit high frequency vibrations and require a time step on the order of 0.5 fs in order to conserve energy. These high frequency modes also limit the outer time step sizes since the modes are coupled. It is therefore desirable to use SHAKE with respa in order to freeze out these high frequency motions and increase the size of the time steps in the respa hierarchy. The following settings can be used for biomolecular simulations with SHAKE and rRESPA:

```
fix             2 all shake 0.000001 500 0 m 1.0 a 1
timestep          4.0
run_style       respa 2 2 inner 1 4.0 5.0 outer 2
```

With these settings, users can expect good energy conservation and roughly a 1.5 fold speedup over the *verlet* style with SHAKE and a 2.0 fs timestep.

For non-biomolecular simulations, the *respa* style can be advantageous if there is a clear separation of time scales - fast and slow modes in the simulation. For example, a system of slowly-moving charged polymer chains could be setup as follows:

```
timestep 4.0
run_style respa 2 8
```

This is two-level rRESPA with an 8x difference between the short and long timesteps. The bonds, angles, dihedrals will be computed every 0.5 fs (assuming real units), while the pair and kspace interactions will be computed once every 4 fs. These are the default settings for each kind of interaction, so no additional keywords are necessary.

Even a LJ system can benefit from rRESPA if the interactions are divided by the inner, middle and outer keywords. A 2-fold or more speedup can be obtained while maintaining good energy conservation. In real units, for a pure LJ fluid at liquid density, with a sigma of 3.0 Angstroms, and epsilon of 0.1 kcal/mol, the following settings seem to work well:

```
timestep  36.0
run_style respa 3 3 4 inner 1 3.0 4.0 middle 2 6.0 7.0 outer 3
```

The *respa/omp* style is a variant of *respa* adapted for use with pair, bond, angle, dihedral, improper, or kspace styles with an *omp* suffix. It is functionally equivalent to *respa* but performs additional operations required for managing *omp* styles. For more on *omp* styles see the *Speed omp* doc page. Accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

You can specify *respa/omp* explicitly in your input script, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

1.95.4 Restrictions

The *verlet/split* style can only be used if LAMMPS was built with the REPLICA package. Correspondingly the *respa/omp* style is available only if the OPENMP package was included. See the *Build package* page for more info.

Run style *verlet/split* is not compatible with kspace styles from the INTEL package and it is not compatible with any tip4p, dipole, or spin kspace styles.

Whenever using rRESPA, the user should experiment with trade-offs in speed and accuracy for their system, and verify that they are conserving energy to adequate precision.

1.95.5 Related commands

timestep, *run*

1.95.6 Default

```
run_style verlet
```

For run_style respa, the default assignment of interactions to rRESPA levels is as follows:

- bond forces = level 1 (innermost loop)
- angle forces = same level as bond forces
- dihedral forces = same level as angle forces
- improper forces = same level as dihedral forces
- pair forces = level N (outermost level)
- kspace forces = same level as pair forces
- inner, middle, outer forces = no default

(Tuckerman) Tuckerman, Berne and Martyna, J Chem Phys, 97, p 1990 (1992).

1.96 set command

1.96.1 Syntax

```
set style ID keyword values ...
```

- style = *atom* or *type* or *mol* or *group* or *region*
- ID = depends on style

for style = atom, ID = a range of atom IDs
for style = type, ID = a range of numeric types or a single type label
for style = mol, ID = a range of molecule IDs
for style = group, ID = a group ID
for style = region, ID = a region ID

- one or more keyword/value pairs may be appended
- keyword = *type* or *type/fraction* or *type/ratio* or *type/subset* or *mol* or *x* or *y* or *z* or *vx* or *vy* or *vz* or *charge* or *dipole* or *dipole/random* or *quat* or *spin/atom* or *spin/atom/random* or *spin/electron* or *radius/electron* or *quat* or *quat/random* or *diameter* or *shape* or *length* or *tri* or *theta* or *theta/random* or *angmom* or *omega* or *mass* or *density* or *density/disc* or *temperature* or *volume* or *image* or *bond* or *angle* or *dihedral* or *improper* or *sph/e* or *sph/cv* or *sph/rho* or *smd/contact/radius* or *smd/mass/density* or *dpd/theta* or *edpd/temp* or *edpd/cv* or *cc* or *epsilon* or *i_name* or *d_name* or *i2_name* or *d2_name*

  type value = numeric atom type or type label
    value can be an atom-style variable (see below)
  type/fraction values = type fraction seed
    type = numeric atom type or type label

fraction = approximate fraction of selected atoms to set to new atom type
seed = random # seed (positive integer)
type/ratio values = type fraction seed
type = numeric atom type or type label
fraction = exact fraction of selected atoms to set to new atom type
seed = random # seed (positive integer)
type/subset values = type Nsubset seed
type = numeric atom type or type label
Nsubset = exact number of selected atoms to set to new atom type
seed = random # seed (positive integer)
mol value = molecule ID
value can be an atom-style variable (see below)
x,y,z value = atom coordinate (distance units)
value can be an atom-style variable (see below)
vx,vy,vz value = atom velocity (velocity units)
value can be an atom-style variable (see below)
charge value = atomic charge (charge units)
value can be an atom-style variable (see below)
dipole values = x y z
x,y,z = orientation of dipole moment vector
any of x,y,z can be an atom-style variable (see below)
dipole/random value = seed Dlen
seed = random # seed (positive integer) for dipole moment orientations
Dlen = magnitude of dipole moment (dipole units)
spin/atom values = g x y z
g = magnitude of magnetic spin vector (in Bohr magneton's unit)
x,y,z = orientation of magnetic spin vector
any of x,y,z can be an atom-style variable (see below)
spin/atom/random value = seed Dlen
seed = random # seed (positive integer) for magnetic spin orientations
Dlen = magnitude of magnetic spin vector (in Bohr magneton's unit)
radius/electron values = eradius
eradius = electron radius (or fixed-core radius) (distance units)
spin/electron value = espin
espin = electron spin (+1/-1), 0 = nuclei, 2 = fixed-core, 3 = pseudo-cores (i.e. ECP)
quat values = a b c theta
a,b,c = unit vector to rotate particle around via right-hand rule
theta = rotation angle (degrees)
any of a,b,c,theta can be an atom-style variable (see below)
quat/random value = seed
seed = random # seed (positive integer) for quaternion orientations
diameter value = diameter of spherical particle (distance units)
value can be an atom-style variable (see below)
shape value = Sx Sy Sz
Sx,Sy,Sz = 3 diameters of ellipsoid (distance units)
length value = len
len = length of line segment (distance units)
len can be an atom-style variable (see below)
tri value = side
side = side length of equilateral triangle (distance units)
side can be an atom-style variable (see below)
theta value = angle (degrees)
angle = orientation of line segment with respect to x-axis
angle can be an atom-style variable (see below)

theta/random value = seed
  seed = random # seed (positive integer) for line segment orienations
angmom values = Lx Ly Lz
  Lx,Ly,Lz = components of angular momentum vector (distance-mass-velocity units)
  any of Lx,Ly,Lz can be an atom-style variable (see below)
omega values = Wx Wy Wz
  Wx,Wy,Wz = components of angular velocity vector (radians/time units)
  any of wx,wy,wz can be an atom-style variable (see below)
mass value = per-atom mass (mass units)
  value can be an atom-style variable (see below)
density value = particle density for a sphere or ellipsoid (mass/distance^3 units), or for a triangle (mass/distance^2 units) or line (mass/distance units) particle
  value can be an atom-style variable (see below)
density/disc value = particle density for a 2d disc or ellipse (mass/distance^2 units)
  value can be an atom-style variable (see below)
temperature value = temperature for finite-size particles (temperature units)
  value can be an atom-style variable (see below)
volume value = particle volume for Peridynamic particle (distance^3 units)
  value can be an atom-style variable (see below)
image nx ny nz
  nx,ny,nz = which periodic image of the simulation box the atom is in
  any of nx,ny,nz can be an atom-style variable (see below)
bond value = numeric bond type or bond type label, for all bonds between selected atoms
angle value = numeric angle type or angle type label, for all angles between selected atoms
dihedral value = numeric dihedral type or dihedral type label, for all dihedrals between selected atoms
improper value = numeric improper type or improper type label, for all impropers between selected atoms
rheo/rho value = density of RHEO particles (mass/distance^3)
rheo/status value = status or phase of RHEO particles (unitless)
sph/e value = energy of SPH particles (need units)
  value can be an atom-style variable (see below)
sph/cv value = heat capacity of SPH particles (need units)
  value can be an atom-style variable (see below)
sph/rho value = density of SPH particles (need units)
  value can be an atom-style variable (see below)
smd/contact/radius = radius for short range interactions, i.e. contact and friction
  value can be an atom-style variable (see below)
smd/mass/density = set particle mass based on volume by providing a mass density
  value can be an atom-style variable (see below)
dpd/theta value = internal temperature of DPD particles (temperature units)
  value can be an atom-style variable (see below)
  value can be NULL which sets internal temp of each particle to KE temp
edpd/temp value = temperature of eDPD particles (temperature units)
  value can be an atom-style variable (see below)
edpd/cv value = volumetric heat capacity of eDPD particles (energy/temperature/volume units)
  value can be an atom-style variable (see below)
cc values = index cc
  index = index of a chemical species (1 to Nspecies)
  cc = chemical concentration of tDPD particles for a species (mole/volume units)
epsilon value = dielectric constant of the medium where the atoms reside
i_name value = custom integer vector with name
d_name value = custom floating-point vector with name
i2_name value = column of a custom integer array with name

column specified as i2_name[N] where N is 1 to Ncol
d2_name value = column of a custom floating-point array with name
column specified as d2_name[N] where N is 1 to Ncol

1.96.2 Examples

```
set group solvent type 2
set group solvent type C
set group solvent type/fraction 2 0.5 12393
set group solvent type/fraction C 0.5 12393
set group edge bond 4
set region half charge 0.5
set type 3 charge 0.5
set type H charge 0.5
set type 1*3 charge 0.5
set atom * charge v_atomfile
set atom 100*200 x 0.5 y 1.0
set atom 100 vx 0.0 vy 0.0 vz -1.0
set atom 1492 type 3
set atom 1492 type H
set atom * i_myVal 5
set atom * d2_Sxyz[1] 6.4
```

1.96.3 Description

Set one or more properties of one or more atoms. Since atom properties are initially assigned by the *read_data*, *read_restart* or *create_atoms* commands, this command changes those assignments. This can be useful for overriding the default values assigned by the *create_atoms* command (e.g. charge = 0.0). It can be useful for altering pairwise and molecular force interactions, since force-field coefficients are defined in terms of types. It can be used to change the labeling of atoms by atom type or molecule ID when they are output in *dump* files. It can also be useful for debugging purposes; i.e. positioning an atom at a precise location to compute subsequent forces or energy.

Note that the *style* and *ID* arguments determine which atoms have their properties reset. The remaining keywords specify which properties to reset and what the new values are. Some strings like *type* or *mol* can be used as a style and/or a keyword.

This section describes how to select which atoms to change the properties of, via the *style* and *ID* arguments.

Changed in version 28Mar2023: Support for type labels was added for selecting atoms by type

The style *atom* selects all the atoms in a range of atom IDs.

The style *type* selects all the atoms in a range of types or type labels. The style *type* selects atoms in one of two ways. A range of numeric atom types can be specified. Or a single atom type label can be specified, e.g. "C". The style *mol* selects all the atoms in a range of molecule IDs.

In each of the range cases, the range can be specified as a single numeric value, or a wildcard asterisk can be used to specify a range of values. This takes the form "*" or "*n" or "n*" or "m*n". For example, for the style *type*, if N = the number of atom types, then an asterisk with no numeric values means all types from 1 to N. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to N (inclusive). A middle asterisk means all types from m to n (inclusive). For all the styles except *mol*, the lowest value for the wildcard is 1; for *mol* it is 0.

The style *group* selects all the atoms in the specified group. The style *region* selects all the atoms in the specified geometric region. See the *group* and *region* commands for details of how to specify a group or region.

This section describes the keyword options for which properties to change, for the selected atoms.

Note that except where explicitly prohibited below, all of the keywords allow an *atom-style or atomfile-style variable* to be used as the specified value(s). If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated, and its resulting per-atom value used to determine the value assigned to each selected atom. Note that the per-atom value from the variable will be ignored for atoms that are not selected via the *style* and *ID* settings explained above. A simple way to use per-atom values from the variable to reset a property for all atoms is to use style *atom* with *ID* = "*"; this selects all atom IDs.

Atom-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. They can also include per-atom values, such as atom coordinates. Thus it is easy to specify a time-dependent or spatially-dependent set of per-atom values. As explained on the *variable* doc page, atomfile-style variables can be used in place of atom-style variables, and thus as arguments to the set command. Atomfile-style variables read their per-atoms values from a file.

Note

Atom-style and atomfile-style variables return floating point per-atom values. If the values are assigned to an integer variable, such as the molecule ID, then the floating point value is truncated to its integer portion, e.g. a value of 2.6 would become 2.

Changed in version 28Mar2023: Support for type labels was added for setting atom, bond, angle, dihedral, and improper types

Keyword *type* sets the atom type for all selected atoms. A specified value can be either a numeric atom type or an atom type label. When using a numeric type, the specified value must be from 1 to ntypes, where ntypes was set by the *create_box* command or the *atom types* field in the header of the data file read by the *read_data* command. When using a type label it must have been defined previously. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

Keyword *type/fraction* sets the atom type for a fraction of the selected atoms. The actual number of atoms changed is not guaranteed to be exactly the specified fraction (0 <= *fraction* <= 1), but should be statistically close. Random numbers are used in such a way that a particular atom is changed or not changed, regardless of how many processors are being used. This keyword does not allow use of an atom-style variable.

Keywords *type/ratio* and *type/subset* also set the atom type for a fraction of the selected atoms. The actual number of atoms changed will be exactly the requested number. For *type/ratio* the specified fraction (0 <= *fraction* <= 1) determines the number. For *type/subset*, the specified *Nsubset* is the number. An iterative algorithm is used which ensures the correct number of atoms are selected, in a perfectly random fashion. Which atoms are selected will change with the number of processors used. These keywords do not allow use of an atom-style variable.

Keyword *mol* sets the molecule ID for all selected atoms. The *atom style* being used must support the use of molecule IDs.

Keywords *x*, *y*, *z*, and *charge* set the coordinates or charge of all selected atoms. For *charge*, the *atom style* being used must support the use of atomic charge. Keywords *vx*, *vy*, and *vz* set the velocities of all selected atoms.

Keyword *dipole* uses the specified x,y,z values as components of a vector to set as the orientation of the dipole moment vectors of the selected atoms. The magnitude of the dipole moment is set by the length of this orientation vector.

Keyword *dipole/random* randomizes the orientation of the dipole moment vectors for the selected atoms and sets the magnitude of each to the specified *Dlen* value. For 2d systems, the z component of the orientation is set to 0.0. Random numbers are used in such a way that the orientation of a particular atom is the same, regardless of how many processors are being used. This keyword does not allow use of an atom-style variable.

Changed in version 15Sep2022.

Keyword *spin/atom* uses the specified g value to set the magnitude of the magnetic spin vectors, and the x,y,z values as components of a vector to set as the orientation of the magnetic spin vectors of the selected atoms. This keyword was previously called *spin*.

Changed in version 15Sep2022.

Keyword *spin/atom/random* randomizes the orientation of the magnetic spin vectors for the selected atoms and sets the magnitude of each to the specified *Dlen* value. This keyword was previously called *spin/random*.

Added in version 15Sep2022.

Keyword *radius/electron* uses the specified value to set the radius of electrons or fixed cores.

Added in version 15Sep2022.

Keyword *spin/electron* sets the spin of an electron (+/- 1) or indicates nuclei (=0), fixed-cores (=2), or pseudo-cores (= 3).

Keyword *quat* uses the specified values to create a quaternion (4-vector) that represents the orientation of the selected atoms. The particles must define a quaternion for their orientation (e.g. ellipsoids, triangles, body particles) as defined by the *atom_style* command. Note that particles defined by *atom_style ellipsoid* have 3 shape parameters. The 3 values must be non-zero for each particle set by this command. They are used to specify the aspect ratios of an ellipsoidal particle, which is oriented by default with its x-axis along the simulation box's x-axis, and similarly for y and z. If this body is rotated (via the right-hand rule) by an angle theta around a unit rotation vector (a,b,c), then the quaternion that represents its new orientation is given by (cos(theta/2), a*sin(theta/2), b*sin(theta/2), c*sin(theta/2)). The theta and a,b,c values are the arguments to the *quat* keyword. LAMMPS normalizes the quaternion in case (a,b,c) was not specified as a unit vector. For 2d systems, the a,b,c values are ignored, since a rotation vector of (0,0,1) is the only valid choice.

Keyword *quat/random* randomizes the orientation of the quaternion for the selected atoms. The particles must define a quaternion for their orientation (e.g. ellipsoids, triangles, body particles) as defined by the *atom_style* command. Random numbers are used in such a way that the orientation of a particular atom is the same, regardless of how many processors are being used. For 2d systems, only orientations in the xy plane are generated. As with keyword *quat*, for ellipsoidal particles, the 3 shape values must be non-zero for each particle set by this command. This keyword does not allow use of an atom-style variable.

Keyword *diameter* sets the size of the selected atoms. The particles must be finite-size spheres as defined by the *atom_style sphere* command. The diameter of a particle can be set to 0.0, which means they will be treated as point particles. Note that this command does not adjust the particle mass, even if it was defined with a density, e.g. via the *read_data* command.

Keyword *shape* sets the size and shape of the selected atoms. The particles must be ellipsoids as defined by the *atom_style ellipsoid* command. The *Sx*, *Sy*, *Sz* settings are the 3 diameters of the ellipsoid in each direction. All 3 can be set to the same value, which means the ellipsoid is effectively a sphere. They can also all be set to 0.0 which means the particle will be treated as a point particle. Note that this command does not adjust the particle mass, even if it was defined with a density, e.g. via the *read_data* command.

Keyword *length* sets the length of selected atoms. The particles must be line segments as defined by the *atom_style line* command. If the specified value is non-zero the line segment is (re)set to a length = the specified value, centered around the particle position, with an orientation along the x-axis. If the specified value is 0.0, the particle will become a point particle. Note that this command does not adjust the particle mass, even if it was defined with a density, e.g. via the *read_data* command.

Keyword *tri* sets the size of selected atoms. The particles must be triangles as defined by the *atom_style tri* command. If the specified value is non-zero the triangle is (re)set to be an equilateral triangle in the xy plane with side length = the specified value, with a centroid at the particle position, with its base parallel to the x axis, and the y-axis running from the center of the base to the top point of the triangle. If the specified value is 0.0, the particle will become a point particle. Note that this command does not adjust the particle mass, even if it was defined with a density, e.g. via the *read_data* command.

Keyword *theta* sets the orientation of selected atoms. The particles must be line segments as defined by the *atom_style line* command. The specified value is used to set the orientation angle of the line segments with respect to the x axis.

Keyword *theta/random* randomizes the orientation of theta for the selected atoms. The particles must be line segments as defined by the *atom_style line* command. Random numbers are used in such a way that the orientation of a particular atom is the same, regardless of how many processors are being used. This keyword does not allow use of an atom-style variable.

Keyword *angmom* sets the angular momentum of selected atoms. The particles must be ellipsoids as defined by the *atom_style ellipsoid* command or triangles as defined by the *atom_style tri* command. The angular momentum vector of the particles is set to the 3 specified components.

Keyword *omega* sets the angular velocity of selected atoms. The particles must be spheres as defined by the *atom_style sphere* command. The angular velocity vector of the particles is set to the 3 specified components.

Keyword *mass* sets the mass of all selected particles. The particles must have a per-atom mass attribute, as defined by the *atom_style* command. See the "mass" command for how to set mass values on a per-type basis.

Keyword *density* or *density/disc* also sets the mass of all selected particles, but in a different way. The particles must have a per-atom mass attribute, as defined by the *atom_style* command. If the atom has a radius attribute (see *atom_style sphere*) and its radius is non-zero, its mass is set from the density and particle volume for 3d systems (the input density is assumed to be in mass/distance^3 units). For 2d, the default is for LAMMPS to model particles with a radius attribute as spheres. However, if the *density/disc* keyword is used, then they can be modeled as 2d discs (circles). Their mass is set from the density and particle area (the input density is assumed to be in mass/distance^2 units).

If the atom has a shape attribute (see *atom_style ellipsoid*) and its 3 shape parameters are non-zero, then its mass is set from the density and particle volume (the input density is assumed to be in mass/distance^3 units). The *density/disc* keyword has no effect; it does not (yet) treat 3d ellipsoids as 2d ellipses.

If the atom has a length attribute (see *atom_style line*) and its length is non-zero, then its mass is set from the density and line segment length (the input density is assumed to be in mass/distance units). If the atom has an area attribute (see *atom_style tri*) and its area is non-zero, then its mass is set from the density and triangle area (the input density is assumed to be in mass/distance^2 units).

If none of these cases are valid, then the mass is set to the density value directly (the input density is assumed to be in mass units).

Keyword *temperature* sets the temperature of a finite-size particle. Currently, only the GRANULAR package supports this attribute. The temperature must be added using an instance of *fix property/atom* The values for the temperature must be positive.

Keyword *volume* sets the volume of all selected particles. Currently, only the *atom_style peri* command defines particles with a volume attribute. Note that this command does not adjust the particle mass.

Keyword *image* sets which image of the simulation box the atom is considered to be in. An image of 0 means it is inside the box as defined. A value of 2 means add 2 box lengths to get the true value. A value of -1 means subtract 1 box length to get the true value. LAMMPS updates these flags as atoms cross periodic boundaries during the simulation. The flags can be output with atom snapshots via the *dump* command. If a value of NULL is specified for any of nx,ny,nz, then the current image value for that dimension is unchanged. For non-periodic dimensions only a value of 0 can be specified. This command can be useful after a system has been equilibrated and atoms have diffused one or more box lengths in various directions. This command can then reset the image values for atoms so that they are effectively inside the simulation box, e.g if a diffusion coefficient is about to be measured via the *compute msd* command. Care should be taken not to reset the image flags of two atoms in a bond to the same value if the bond straddles a periodic boundary (rather they should be different by +/- 1). This will not affect the dynamics of a simulation, but may mess up analysis of the trajectories if a LAMMPS diagnostic or your own analysis relies on the image flags to unwrap a molecule which straddles the periodic box.

Keywords *bond*, *angle*, *dihedral*, and *improper*, set the bond type (angle type, etc) of all bonds (angles, etc) of selected atoms to the specified value. The value can be a numeric type from 1 to nbondtypes (nangletypes, etc). Or it can be a type label (bond type label, angle type label, etc). See the *Howto type labels* doc page for the allowed syntax of type

labels and a general discussion of how type labels can be used. All atoms in a particular bond (angle, etc) must be selected atoms in order for the change to be made. The value of nbondtypes (nangletypes, etc) was set by the *bond types* (*angle types*, etc) field in the header of the data file read by the *read_data* command. These keywords do not allow use of an atom-style variable.

Keywords *rheo/rho* and *rheo/status* set the density and the status of rheo particles. In particular, one can only set the phase in the status as described by the *RHEO howto page*.

Keywords *sph/e*, *sph/cv*, and *sph/rho* set the energy, heat capacity, and density of smoothed particle hydrodynamics (SPH) particles. See this PDF guide to using SPH in LAMMPS.

Keyword *smd/mass/density* sets the mass of all selected particles, but it is only applicable to the Smooth Mach Dynamics package MACHDYN. It assumes that the particle volume has already been correctly set and calculates particle mass from the provided mass density value.

Keyword *smd/contact/radius* only applies to simulations with the Smooth Mach Dynamics package MACHDYN. Itsets an interaction radius for computing short-range interactions, e.g. repulsive forces to prevent different individual physical bodies from penetrating each other. Note that the SPH smoothing kernel diameter used for computing long range, nonlocal interactions, is set using the *diameter* keyword.

Keyword *dpd/theta* sets the internal temperature of a DPD particle as defined by the DPD-REACT package. If the specified value is a number it must be >= 0.0. If the specified value is NULL, then the kinetic temperature Tkin of each particle is computed as 3/2 k Tkin = KE = 1/2 m v^2 = 1/2 m (vx*vx+vy*vy+vz*vz). Each particle's internal temperature is set to Tkin. If the specified value is an atom-style variable, then the variable is evaluated for each particle. If a value >= 0.0, the internal temperature is set to that value. If it is < 0.0, the computation of Tkin is performed and the internal temperature is set to that value.

Keywords *edpd/temp* and *edpd/cv* set the temperature and volumetric heat capacity of an eDPD particle as defined by the DPD-MESO package. Currently, only *atom_style edpd* defines particles with these attributes. The values for the temperature and heat capacity must be positive.

Keyword *cc* sets the chemical concentration of a tDPD particle for a specified species as defined by the DPD-MESO package. Currently, only *atom_style tdpd* defines particles with this attribute. An integer for "index" selects a chemical species (1 to Nspecies) where Nspecies is set by the atom_style command. The value for the chemical concentration must be >= 0.0.

Keyword *epsilon* sets the dielectric constant of a particle, precisely of the medium where the particle resides as defined by the DIELECTRIC package. Currently, only *atom_style dielectric* defines particles with this attribute. The value for the dielectric constant must be >= 0.0. Note that the set command with this keyword will rescale the particle charge accordingly so that the real charge (e.g., as read from a data file) stays intact. To change the real charges, one needs to use the set command with the *charge* keyword. Care must be taken to ensure that the real and scaled charges, and dielectric constants are consistent.

Keywords *i_name*, *d_name*, *i2_name*, *d2_name* refer to custom per-atom integer and floating-point vectors or arrays that have been added via the *fix property/atom* command. When that command is used specific names are given to each attribute which are the "name" portion of these keywords. For arrays *i2_name* and *d2_name*, the column of the array must also be included following the name in brackets: e.g. d2_xyz[2], i2_mySpin[3].

1.96.4 Restrictions

You cannot set an atom attribute (e.g. *mol* or *q* or *volume*) if the *atom_style* does not have that attribute.

This command requires inter-processor communication to coordinate the setting of bond types (angle types, etc). This means that your system must be ready to perform a simulation before using one of these keywords (force fields set, atom mass set, etc). This is not necessary for other keywords.

Using the *region* style with the bond (angle, etc) keywords can give unpredictable results if there are bonds (angles, etc) that straddle periodic boundaries. This is because the region may only extend up to the boundary and partner atoms in the bond (angle, etc) may have coordinates outside the simulation box if they are ghost atoms.

1.96.5 Related commands

create_box, *create_atoms*, *read_data*

1.96.6 Default

none

1.97 shell command

1.97.1 Syntax

```
shell command args
```

- command = *cd* or *mkdir* or *mv* or *rm* or *rmdir* or *putenv* or arbitrary command

  cd arg = dir
    dir = directory to change to
  mkdir args = dir1 dir2 ...
    dir1,dir2 = one or more directories to create
  mv args = old new
    old = old filename
    new = new filename or destination folder
  rm args = [-f] file1 file2 ...
    -f = turn off warnings (optional)
    file1,file2 = one or more filenames to delete
  rmdir args = dir1 dir2 ...
    dir1,dir2 = one or more directories to delete
  putenv args = var1=value1 var2=value2
    var=value = one of more definitions of environment variables
  anything else is passed as a command to the shell for direct execution

1.97.2 Examples

```
shell cd sub1
shell cd ..
shell mkdir tmp1 tmp2/tmp3
shell rmdir tmp1 tmp2
shell mv log.lammps hold/log.1
shell rm TMP/file1 TMP/file2
shell putenv LAMMPS_POTENTIALS=../../potentials
shell my_setup file1 10 file2
shell my_post_process 100 dump.out
```

1.97.3 Description

Execute a shell command. A few simple file-based shell commands are supported directly, in Unix-style syntax. Any command not listed above is passed as-is to the C-library system() call, which invokes the command in a shell. To use the external executable instead of the built-in version one needs to use a full path, for example */bin/rm* instead of *rm*. The built-in commands will also work on operating systems, that do not - by default - provide the corresponding external executables (like *mkdir* on Windows).

This command provides a ways to invoke custom commands or executables from your input script. For example, you can move files around in preparation for the next section of the input script. Or you can run a program that pre-processes data for input into LAMMPS. Or you can run a program that post-processes LAMMPS output data.

With the exception of *cd*, all commands, including ones invoked via a system() call, are executed by only a single processor, so that files/directories are not being manipulated by multiple processors concurrently which may result in unexpected errors or corrupted files.

The *cd* command changes the current working directory similar to the cd command. All subsequent LAMMPS commands that read/write files will use the new directory. All processors execute this command.

The *mkdir* command creates directories similar to the Unix mkdir -p command. That is, it will attempt to create the entire path of subdirectories if they do not exist yet.

The *mv* command renames a file and/or moves it to a new directory. It cannot rename files across filesystem boundaries or between drives.

The *rm* command deletes file similar to the Unix rm command.

The *rmdir* command deletes directories similar to Unix rmdir command. If a directory is not empty, its contents are also removed recursively similar to the Unix rm -r command.

The *putenv* command defines or updates an environment variable directly. Since this command does not pass through the shell, no shell variable expansion or globbing is performed, only the usual substitution for LAMMPS variables defined with the *variable* command is performed. The resulting string is then used literally.

Any other command is passed as-is to the shell along with its arguments as one string, invoked by the C-library system() call. For example, these lines in your input script:

```
variable n equal 10
variable foo string file2
shell my_setup file1 $n ${foo}
```

would be the same as invoking

```
my_setup file1 10 file2
```

from a command-line prompt. The executable program "my_setup" is run with 3 arguments: file1 10 file2.

1.97.4 Restrictions

LAMMPS will do a best effort to detect errors and print suitable warnings, but due to the nature of delegating commands to the C-library system() call, this is not always reliable.

1.97.5 Related commands

none

1.97.6 Default

none

1.98 special_bonds command

1.98.1 Syntax

```
special_bonds keyword values ...
```

- one or more keyword/value pairs may be appended
- keyword = *amber* or *charmm* or *dreiding* or *fene* or *lj/coul* or *lj* or *coul* or *angle* or *dihedral* or *one/five*

  amber values = none
  charmm values = none
  dreiding values = none
  fene values = none
  lj/coul values = w1,w2,w3
    w1,w2,w3 = weights (0.0 to 1.0) on pairwise Lennard-Jones and Coulombic interactions
  lj values = w1,w2,w3
    w1,w2,w3 = weights (0.0 to 1.0) on pairwise Lennard-Jones interactions
  coul values = w1,w2,w3
    w1,w2,w3 = weights (0.0 to 1.0) on pairwise Coulombic interactions
  angle value = yes or no
  dihedral value = yes or no
  one/five value = yes or no

1.98.2 Examples

```
special_bonds amber
special_bonds charmm
special_bonds fene dihedral no
special_bonds lj/coul 0.0 0.0 0.5 angle yes dihedral yes
special_bonds lj 0.0 0.0 0.5 coul 0.0 0.0 0.0 dihedral yes
```

1.98.3 Description

Set weighting coefficients for pairwise energy and force contributions between pairs of atoms that are also permanently bonded to each other, either directly or via one or two intermediate bonds. These weighting factors are used by nearly all *pair styles* in LAMMPS that compute simple pairwise interactions. Permanent bonds between atoms are specified by defining the bond topology in the data file read by the *read_data* command. Typically a *bond_style* command is also used to define a bond potential. The rationale for using these weighting factors is that the interaction between a pair of bonded atoms is all (or mostly) specified by the bond, angle, dihedral potentials, and thus the non-bonded Lennard-Jones or Coulombic interaction between the pair of atoms should be excluded (or reduced by a weighting factor).

Note

These weighting factors are NOT used by *pair styles* that compute many-body interactions, since the "bonds" that result from such interactions are not permanent, but are created and broken dynamically as atom conformations change. Examples of pair styles in this category are EAM, MEAM, Stillinger-Weber, Tersoff, COMB, AIREBO, and ReaxFF. In fact, it generally makes no sense to define permanent bonds between atoms that interact via these potentials, though such bonds may exist elsewhere in your system, e.g. when using the *pair_style hybrid* command. Thus LAMMPS ignores special_bonds settings when many-body potentials are calculated. Please note, that the existence of explicit bonds for atoms that are described by a many-body potential will alter the neighbor list and thus can render the computation of those interactions invalid, since those pairs are not only used to determine direct pairwise interactions but also neighbors of neighbors and more. The recommended course of action is to remove such bonds, or - if that is not possible - use a special bonds setting of 1.0 1.0 1.0.

Note

Unlike some commands in LAMMPS, you cannot use this command multiple times in an incremental fashion: e.g. to first set the LJ settings and then the Coulombic ones. Each time you use this command it sets all the coefficients to default values and only overrides the one you specify, so you should set all the options you need each time you use it. See more details at the bottom of this page.

The Coulomb factors are applied to any Coulomb (charge interaction) term that the potential calculates. The LJ factors are applied to the remaining terms that the potential calculates, whether they represent LJ interactions or not. The weighting factors are a scaling prefactor on the energy and force between the pair of atoms.

A value of 1.0 means include the full interaction without flagging the pair as a "special pair"; a value of 0.0 means exclude the pair completely from the neighbor list, except for pair styles that require a *kspace style* and pair styles *amoeba*, *hippo*, *thole*, *coul/exclude*, and pair styles that include "coul/dsf" or "coul/wolf".

Note

To include pairs that would otherwise be excluded (so they are included in the neighbor list for certain analysis compute styles), you can use a very small but non-zero value like 1.0e-100 instead of 0.0. Due to using floating-point math, the computed force, energy, and virial contributions from the pairs will be too small to cause differences.

The first of the 3 coefficients (LJ or Coulombic) is the weighting factor on 1-2 atom pairs, which are pairs of atoms directly bonded to each other. The second coefficient is the weighting factor on 1-3 atom pairs which are those separated by 2 bonds (e.g. the two H atoms in a water molecule). The third coefficient is the weighting factor on 1-4 atom pairs which are those separated by 3 bonds (e.g. the first and fourth atoms in a dihedral interaction). Thus if the 1-2 coefficient is set to 0.0, then the pairwise interaction is effectively turned off for all pairs of atoms bonded to each other. If it is set to 1.0, then that interaction will be at full strength.

Note

For purposes of computing weighted pairwise interactions, 1-3 and 1-4 interactions are not defined from the list of angles or dihedrals used by the simulation. Rather, they are inferred topologically from the set of bonds specified when the simulation is defined from a data or restart file (see *read_data* or *read_restart* commands). Thus the set of 1-2,1-3,1-4 interactions that the weights apply to is the same whether angle and dihedral potentials are computed or not, and remains the same even if bonds are constrained, or turned off, or removed during a simulation.

The two exceptions to this rule are (a) if the *angle* or *dihedral* keywords are set to *yes* (see below), or (b) if the *delete_bonds* command is used with the *special* option that re-computes the 1-2,1-3,1-4 topologies after bonds are deleted; see the *delete_bonds* command for more details.

The *amber* keyword sets the 3 coefficients to 0.0, 0.0, 0.5 for LJ interactions and to 0.0, 0.0, 0.8333 for Coulombic interactions, which is the default for a commonly used version of the AMBER force field, where the last value is really 5/6. See *(Cornell)* for a description of the AMBER force field.

The *charmm* keyword sets the 3 coefficients to 0.0, 0.0, 0.0 for both LJ and Coulombic interactions, which is the default for a commonly used version of the CHARMM force field. Note that in pair styles *lj/charmm/coul/charmm* and *lj/charmm/coul/long* the 1-4 coefficients are defined explicitly, and these pairwise contributions are computed as part of the charmm dihedral style - see the *pair_coeff* and *dihedral_style* commands for more information. See *(MacKerell)* for a description of the CHARMM force field.

The *dreiding* keyword sets the 3 coefficients to 0.0, 0.0, 1.0 for both LJ and Coulombic interactions, which is the default for the Dreiding force field, as discussed in *(Mayo)*.

The *fene* keyword sets the 3 coefficients to 0.0, 1.0, 1.0 for both LJ and Coulombic interactions, which is consistent with a coarse-grained polymer model with *FENE bonds*. See *(Kremer)* for a description of FENE bonds.

The *lj/coul*, *lj*, and *coul* keywords allow the 3 coefficients to be set explicitly. The *lj/coul* keyword sets both the LJ and Coulombic coefficients to the same 3 values. The *lj* and *coul* keywords only set either the LJ or Coulombic coefficients. Use both of them if you wish to set the LJ coefficients to different values than the Coulombic coefficients.

The *angle* keyword allows the 1-3 weighting factor to be ignored for individual atom pairs if they are not listed as the first and last atoms in any angle defined in the simulation or as 1,3 or 2,4 atoms in any dihedral defined in the simulation. For example, imagine the 1-3 weighting factor is set to 0.5 and you have a linear molecule with 4 atoms and bonds as follows: 1-2-3-4. If your data file defines 1-2-3 as an angle, but does not define 2-3-4 as an angle or 1-2-3-4 as a dihedral, then the pairwise interaction between atoms 1 and 3 will always be weighted by 0.5, but different force fields use different rules for weighting the pairwise interaction between atoms 2 and 4. If the *angle* keyword is specified as *yes*, then the pairwise interaction between atoms 2 and 4 will be unaffected (full weighting of 1.0). If the *angle* keyword is specified as *no* which is the default, then the 2,4 interaction will also be weighted by 0.5.

The *dihedral* keyword allows the 1-4 weighting factor to be ignored for individual atom pairs if they are not listed as the first and last atoms in any dihedral defined in the simulation. For example, imagine the 1-4 weighting factor is set to 0.5 and you have a linear molecule with 5 atoms and bonds as follows: 1-2-3-4-5. If your data file defines 1-2-3-4 as a dihedral, but does not define 2-3-4-5 as a dihedral, then the pairwise interaction between atoms 1 and 4 will always be weighted by 0.5, but different force fields use different rules for weighting the pairwise interaction between atoms 2 and 5. If the *dihedral* keyword is specified as *yes*, then the pairwise interaction between atoms 2 and 5 will be unaffected (full weighting of 1.0). If the *dihedral* keyword is specified as *no* which is the default, then the 2,5 interaction will also be weighted by 0.5.

The *one/five* keyword enable calculation and storage of a list of 1-5 neighbors in the molecular topology for each atom. It is required by some pair styles, such as *pair_style amoeba* and *pair_style hippo*.

Note

LAMMPS stores and maintains a data structure with a list of the first, second, and third neighbors of each atom (within the bond topology of the system). If new bonds are created (or molecules added containing atoms with more special neighbors), the size of this list needs to grow. Note that adding a single bond always adds a new first neighbor but may also induce *many* new second and third neighbors, depending on the molecular topology of your system. Using the *extra/special/per/atom* keyword to either *read_data* or *create_box* reserves empty space in the list for this N additional first, second, or third neighbors to be added. If you do not do this, you may get an error when bonds (or molecules) are added.

Note

If you reuse this command in an input script, you should set all the options you need each time. This command cannot be used a second time incrementally. E.g. these two commands:

```
special_bonds lj 0.0 1.0 1.0
special_bonds coul 0.0 0.0 1.0
```

are not the same as

```
special_bonds lj 0.0 1.0 1.0 coul 0.0 0.0 1.0
```

In the first case you end up with (after the second command):

```
LJ: 0.0 0.0 0.0
Coul: 0.0 0.0 1.0
```

while only in the second case do you get the desired settings of:

```
LJ: 0.0 1.0 1.0
Coul: 0.0 0.0 1.0
```

This happens because the LJ (and Coul) settings are reset to their default values before modifying them, each time the *special_bonds* command is issued.

1.98.4 Restrictions

none

1.98.5 Related commands

delete_bonds, *fix bond/create*

1.98.6 Default

All 3 Lennard-Jones and 3 Coulombic weighting coefficients = 0.0, angle = no, dihedral = no.

(Cornell) Cornell, Cieplak, Bayly, Gould, Merz, Ferguson, Spellmeyer, Fox, Caldwell, Kollman, JACS 117, 5179-5197 (1995).

(Kremer) Kremer, Grest, J Chem Phys, 92, 5057 (1990).

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

(Mayo) Mayo, Olfason, Goddard III, J Phys Chem, 94, 8897-8909 (1990).

1.99 suffix command

1.99.1 Syntax

```
suffix style args
```

- style = *off* or *on* or *gpu* or *intel* or *kk* or *omp* or *opt* or *hybrid*
- args = for hybrid style, default suffix to be used and alternative suffix

1.99.2 Examples

```
suffix off
suffix on
suffix gpu
suffix intel
suffix hybrid intel omp
suffix kk
```

1.99.3 Description

This command allows you to use variants of various styles if they exist. In that respect it operates the same as the *-suffix command-line switch*. It also has options to turn off or back on any suffix setting made via the command line.

The specified style can be *gpu*, *intel*, *kk*, *omp*, *opt* or *hybrid*. These refer to optional packages that LAMMPS can be built with, as described on the *Build package* doc page. The "gpu" style corresponds to the GPU package, the "intel" style to the INTEL package, the "kk" style to the KOKKOS package, the "omp" style to the OPENMP package, and the "opt" style to the OPT package.

These are the variants these packages provide:

- GPU = a handful of pair styles and the PPPM kspace_style, optimized to run on one or more GPUs or multicore CPU/GPU nodes
- INTEL = a collection of pair styles and neighbor routines optimized to run in single, mixed, or double precision on CPUs and Intel(R) Xeon Phi(TM) co-processors.
- KOKKOS = a collection of atom, pair, and fix styles optimized to run using the Kokkos library on various kinds of hardware, including GPUs via CUDA and many-core chips via OpenMP or threading.

- OPENMP = a collection of pair, bond, angle, dihedral, improper, kspace, compute, and fix styles with support for OpenMP multi-threading
- OPT = a handful of pair styles, cache-optimized for faster CPU performance
- HYBRID = a combination of two packages can be specified (see below)

As an example, all of the packages provide a *pair_style lj/cut* variant, with style names lj/cut/opt, lj/cut/omp, lj/cut/gpu, lj/cut/intel, or lj/cut/kk. A variant styles can be specified explicitly in your input script, e.g. pair_style lj/cut/gpu. If the suffix command is used with the appropriate style, you do not need to modify your input script. The specified suffix (opt,omp,gpu,intel,kk) is automatically appended whenever your input script command creates a new *atom*, *pair*, *bond*, *angle*, *dihedral*, *improper*, *kspace*, *fix*, *compute*, or *run* style. If the variant version does not exist, the standard version is created.

For "hybrid", two packages are specified. The first is used whenever available. If a style with the first suffix is not available, the style with the suffix for the second package will be used if available. For example, "hybrid intel omp" will use styles from the INTEL package as a first choice and styles from the OPENMP package as a second choice if no INTEL variant is available.

If the specified style is *off*, then any previously specified suffix is temporarily disabled, whether it was specified by a command-line switch or a previous suffix command. If the specified style is *on*, a disabled suffix is turned back on. The use of these 2 commands lets your input script use a standard LAMMPS style (i.e. a non-accelerated variant), which can be useful for testing or benchmarking purposes. Of course this is also possible by not using any suffix commands, and explicitly appending or not appending the suffix to the relevant commands in your input script.

Note

The default *run_style* verlet is invoked prior to reading the input script and is therefore not affected by a suffix command in the input script. The KOKKOS package requires "run_style verlet/kk", so when using the KOKKOS package it is necessary to either use the command line "-sf kk" command or add an explicit "run_style verlet" command to the input script.

1.99.4 Restrictions

none

1.99.5 Related commands

-suffix command-line switch

1.99.6 Default

none

1.100 tad command

1.100.1 Syntax

```
tad N t_event T_lo T_hi delta tmax compute-ID keyword value ...
```

- N = # of timesteps to run (not including dephasing/quenching)
- t_event = timestep interval between event checks
- T_lo = temperature at which event times are desired
- T_hi = temperature at which MD simulation is performed
- delta = desired confidence level for stopping criterion
- tmax = reciprocal of lowest expected pre-exponential factor (time units)
- compute-ID = ID of the compute used for event detection
- zero or more keyword/value pairs may be appended
- keyword = *min* or *neb* or *neb_style* or *neb_step* or *neb_log*

  min values = etol ftol maxiter maxeval
    etol = stopping tolerance for energy (energy units)
    ftol = stopping tolerance for force (force units)
    maxiter = max iterations of minimize
    maxeval = max number of force/energy evaluations
  neb values = ftol N1 N2 Nevery
    etol = stopping tolerance for energy (energy units)
    ftol = stopping tolerance for force (force units)
    N1 = max # of iterations (timesteps) to run initial NEB
    N2 = max # of iterations (timesteps) to run barrier-climbing NEB
    Nevery = print NEB statistics every this many timesteps
  neb_style value = quickmin or fire
  neb_step value = dtneb
    dtneb = timestep for NEB damped dynamics minimization
  neb_log value = file where NEB statistics are printed

1.100.2 Examples

```
tad 2000 50 1800 2300 0.01 0.01 event
tad 2000 50 1800 2300 0.01 0.01 event &
    min 1e-05 1e-05 100 100 &
    neb 0.0 0.01 200 200 20 &
    min_style cg &
    neb_style fire &
    neb_log log.neb
```

1.100.3 Description

Run a temperature accelerated dynamics (TAD) simulation. This method requires two or more partitions to perform NEB transition state searches.

TAD is described in *this paper* by Art Voter. It is a method that uses accelerated dynamics at an elevated temperature to generate results at a specified lower temperature. A good overview of accelerated dynamics methods (AMD) for such systems is given in *this review paper* from the same group. To quote from the review paper: "The dynamical evolution is characterized by vibrational excursions within a potential basin, punctuated by occasional transitions between basins. The transition probability is characterized by p(t) = k*exp(-kt) where k is the rate constant."

TAD is a suitable AMD method for infrequent-event systems, where in addition, the transition kinetics are well-approximated by harmonic transition state theory (hTST). In hTST, the temperature dependence of transition rates follows the Arrhenius relation. As a consequence a set of event times generated in a high-temperature simulation can be mapped to a set of much longer estimated times in the low-temperature system. However, because this mapping involves the energy barrier of the transition event, which is different for each event, the first event at the high temperature may not be the earliest event at the low temperature. TAD handles this by first generating a set of possible events from the current basin. After each event, the simulation is reflected backwards into the current basin. This is repeated until the stopping criterion is satisfied, at which point the event with the earliest low-temperature occurrence time is selected. The stopping criterion is that the confidence measure be greater than 1-*delta*. The confidence measure is the probability that no earlier low-temperature event will occur at some later time in the high-temperature simulation. hTST provides an lower bound for this probability, based on the user-specified minimum pre-exponential factor (reciprocal of *tmax*).

In order to estimate the energy barrier for each event, the TAD method invokes the *NEB* method. Each NEB replica runs on a partition of processors. The current NEB implementation in LAMMPS restricts you to having exactly one processor per replica. For more information, see the documentation for the *neb* command. In the current LAMMPS implementation of TAD, all the non-NEB TAD operations are performed on the first partition, while the other partitions remain idle. See the *Howto replica* doc page for further discussion of multi-replica simulations.

A TAD run has several stages, which are repeated each time an event is performed. The logic for a TAD run is as follows:

```
while (time remains):
  while (time < tstop):
    until (event occurs):
      run dynamics for t_event steps
      quench
    run neb calculation using all replicas
    compute tlo from energy barrier
    update earliest event
    update tstop
    reflect back into current basin
  execute earliest event
```

Before this outer loop begins, the initial potential energy basin is identified by quenching (an energy minimization, see below) the initial state and storing the resulting coordinates for reference.

Inside the inner loop, dynamics is run continuously according to whatever integrator has been specified by the user, stopping every *t_event* steps to check if a transition event has occurred. This check is performed by quenching the system and comparing the resulting atom coordinates to the coordinates from the previous basin.

A quench is an energy minimization and is performed by whichever algorithm has been defined by the *min_style* command; its default is the CG minimizer. The tolerances and limits for each quench can be set by the *min* keyword. Note that typically, you do not need to perform a highly-converged minimization to detect a transition event.

The event check is performed by a compute with the specified *compute-ID*. Currently there is only one compute that works with the TAD command, which is the *compute event/displace* command. Other event-checking computes may

be added. *Compute event/displace* checks whether any atom in the compute group has moved further than a specified threshold distance. If so, an "event" has occurred.

The NEB calculation is similar to that invoked by the *neb* command, except that the final state is generated internally, instead of being read in from a file. The style of minimization performed by NEB is determined by the *neb_style* keyword and must be a damped dynamics minimizer. The tolerances and limits for each NEB calculation can be set by the *neb* keyword. As discussed on the *neb*, it is often advantageous to use a larger timestep for NEB than for normal dynamics. Since the size of the timestep set by the *timestep* command is used by TAD for performing dynamics, there is a *neb_step* keyword which can be used to set a larger timestep for each NEB calculation if desired.

A key aspect of the TAD method is setting the stopping criterion appropriately. If this criterion is too conservative, then many events must be generated before one is finally executed. Conversely, if this criterion is too aggressive, high-entropy high-barrier events will be over-sampled, while low-entropy low-barrier events will be under-sampled. If the lowest pre-exponential factor is known fairly accurately, then it can be used to estimate *tmax*, and the value of *delta* can be set to the desired confidence level e.g. *delta* = 0.05 corresponds to 95% confidence. However, for systems where the dynamics are not well characterized (the most common case), it will be necessary to experiment with the values of *delta* and *tmax* to get a good trade-off between accuracy and performance.

A second key aspect is the choice of *t_hi*. A larger value greatly increases the rate at which new events are generated. However, too large a value introduces errors due to anharmonicity (not accounted for within hTST). Once again, for any given system, experimentation is necessary to determine the best value of *t_hi*.

Five kinds of output can be generated during a TAD run: event statistics, NEB statistics, thermodynamic output by each replica, dump files, and restart files.

Event statistics are printed to the screen and master log.lammps file each time an event is executed. The quantities are the timestep, CPU time, global event number *N*, local event number *M*, event status, energy barrier, time margin, *t_lo* and *delt_lo*. The timestep is the usual LAMMPS timestep, which corresponds to the high-temperature time at which the event was detected, in units of timestep. The CPU time is the total processor time since the start of the TAD run. The global event number *N* is a counter that increments with each executed event. The local event number *M* is a counter that resets to zero upon entering each new basin. The event status is *E* when an event is executed, and is *D* for an event that is detected, while *DF* is for a detected event that is also the earliest (first) event at the low temperature.

The time margin is the ratio of the high temperature time in the current basin to the stopping time. This last number can be used to judge whether the stopping time is too short or too long (see above).

t_lo is the low-temperature event time when the current basin was entered, in units of timestep. del*t_lo* is the time of each detected event, measured relative to *t_lo*. *delt_lo* is equal to the high-temperature time since entering the current basin, scaled by an exponential factor that depends on the hi/lo temperature ratio and the energy barrier for that event.

On lines for executed events, with status *E*, the global event number is incremented by one, the local event number and time margin are reset to zero, while the global event number, energy barrier, and *delt_lo* match the last event with status *DF* in the immediately preceding block of detected events. The low-temperature event time *t_lo* is incremented by *delt_lo*.

NEB statistics are written to the file specified by the *neb_log* keyword. If the keyword value is "none", then no NEB statistics are printed out. The statistics are written every *Nevery* timesteps. See the *neb* command for a full description of the NEB statistics. When invoked from TAD, NEB statistics are never printed to the screen.

Because the NEB calculation must run on multiple partitions, LAMMPS produces additional screen and log files for each partition, e.g. log.lammps.0, log.lammps.1, etc. For the TAD command, these contain the thermodynamic output of each NEB replica. In addition, the log file for the first partition, log.lammps.0, will contain thermodynamic output from short runs and minimizations corresponding to the dynamics and quench operations, as well as a line for each new detected event, as described above.

After the TAD command completes, timing statistics for the TAD run are printed in each replica's log file, giving a breakdown of how much CPU time was spent in each stage (NEB, dynamics, quenching, etc).

Any *dump files* defined in the input script will be written to during a TAD run at timesteps when an event is executed. This means the requested dump frequency in the *dump* command is ignored. There will be one dump file (per dump command) created for all partitions. The atom coordinates of the dump snapshot are those of the minimum energy configuration resulting from quenching following the executed event. The timesteps written into the dump files correspond to the timestep at which the event occurred and NOT the clock. A dump snapshot corresponding to the initial minimum state used for event detection is written to the dump file at the beginning of each TAD run.

If the *restart* command is used, a single restart file for all the partitions is generated, which allows a TAD run to be continued by a new input script in the usual manner. The restart file is generated after an event is executed. The restart file contains a snapshot of the system in the new quenched state, including the event number and the low-temperature time. The restart frequency specified in the *restart* command is interpreted differently when performing a TAD run. It does not mean the timestep interval between restart files. Instead it means an event interval for executed events. Thus a frequency of 1 means write a restart file every time an event is executed. A frequency of 10 means write a restart file every 10th executed event. When an input script reads a restart file from a previous TAD run, the new script can be run on a different number of replicas or processors.

Note that within a single state, the dynamics will typically temporarily continue beyond the event that is ultimately chosen, until the stopping criterion is satisfied. When the event is eventually executed, the timestep counter is reset to the value when the event was detected. Similarly, after each quench and NEB minimization, the timestep counter is reset to the value at the start of the minimization. This means that the timesteps listed in the replica log files do not always increase monotonically. However, the timestep values printed to the master log file, dump files, and restart files are always monotonically increasing.

1.100.4 Restrictions

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

N setting must be integer multiple of *t_event*.

Runs restarted from restart files written during a TAD run will only produce identical results if the user-specified integrator supports exact restarts. So *fix nvt* will produce an exact restart, but *fix langevin* will not.

This command cannot be used when any fixes are defined that keep track of elapsed time to perform time-dependent operations. Examples include the "ave" fixes such as *fix ave/chunk*. Also *fix dt/reset* and *fix deposit*.

1.100.5 Related commands

compute event/displace, *min_modify*, *min_style*, *run_style*, *minimize*, *temper*, *neb*, *prd*

1.100.6 Default

The option defaults are *min* = 0.1 0.1 40 50, *neb* = 0.01 100 100 10, *neb_style* = *quickmin*, *neb_step* = the same timestep set by the *timestep* command, and *neb_log* = "none".

(Voter2000) Sorensen and Voter, J Chem Phys, 112, 9599 (2000)

(Voter2002) Voter, Montalenti, Germann, Annual Review of Materials Research 32, 321 (2002).

1.101 temper command

1.101.1 Syntax

```
temper N M temp fix-ID seed1 seed2 index
```

- N = total # of timesteps to run
- M = attempt a tempering swap every this many steps
- temp = initial temperature for this ensemble
- fix-ID = ID of the fix that will control temperature during the run
- seed1 = random # seed used to decide on adjacent temperature to partner with
- seed2 = random # seed for Boltzmann factor in Metropolis swap
- index = which temperature (0 to N-1) I am simulating (optional)

1.101.2 Examples

```
temper 100000 100 $t tempfix 0 58728
temper 40000 100 $t tempfix 0 32285 $w
```

1.101.3 Description

Run a parallel tempering or replica exchange simulation using multiple replicas (ensembles) of a system. Two or more replicas must be used.

Each replica runs on a partition of one or more processors. Processor partitions are defined at run-time using the *-partition command-line switch*. Note that if you have MPI installed, you can run a multi-replica simulation with more replicas (partitions) than you have physical processors, e.g you can run a 10-replica simulation on one or two processors. You will simply not get the performance speed-up you would see with one or more physical processors per replica. See the *Howto replica* doc page for further discussion.

Each replica's temperature is controlled at a different value by a fix with *fix-ID* that controls temperature. Most thermostat fix styles (with and without included time integration) are supported. The command will print an error message and abort, if the chosen fix is unsupported. The desired temperature is specified by *temp*, which is typically a variable previously set in the input script, so that each partition is assigned a different temperature. See the *variable* command for more details. For example:

```
variable t world 300.0 310.0 320.0 330.0
fix myfix all nvt temp $t $t 100.0
temper 100000 100 $t myfix 3847 58382
```

would define 4 temperatures, and assign one of them to the thermostat used by each replica, and to the temper command.

As the tempering simulation runs for *N* timesteps, a temperature swap between adjacent ensembles will be attempted every *M* timesteps. If *seed1* is 0, then the swap attempts will alternate between odd and even pairings. If *seed1* is non-zero then it is used as a seed in a random number generator to randomly choose an odd or even pairing each time. Each attempted swap of temperatures is either accepted or rejected based on a Boltzmann-weighted Metropolis criterion which uses *seed2* in the random number generator.

As a tempering run proceeds, multiple log files and screen output files are created, one per replica. By default these files are named log.lammps.M and screen.M where M is the replica number from 0 to N-1, with N = # of replicas. See the *-log and -screen command-line swiches* for info on how to change these names.

The main screen and log file (log.lammps) will list information about which temperature is assigned to each replica at each thermodynamic output timestep. E.g. for a simulation with 16 replicas:

```
Running on 16 partitions of processors
Step T0 T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12 T13 T14 T15
0    0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
500  1 0 3 2 5 4 6 7 8 9 10 11 12 13 14 15
1000 2 0 4 1 5 3 6 7 8 9 10 11 12 14 13 15
1500 2 1 4 0 5 3 6 7 9 8 10 11 12 14 13 15
2000 2 1 3 0 6 4 5 7 10 8 9 11 12 14 13 15
2500 2 1 3 0 6 4 5 7 11 8 9 10 12 14 13 15
...
```

The column headings T0 to TN-1 mean which temperature is currently assigned to the replica 0 to N-1. Thus the columns represent replicas and the value in each column is its temperature (also numbered 0 to N-1). For example, a 0 in the fourth column (column T3, step 2500) means that the fourth replica is assigned temperature 0, i.e. the lowest temperature. You can verify this time sequence of temperature assignments for the Nth replica by comparing the Nth column of screen output to the thermodynamic data in the corresponding log.lammps.N or screen.N files as time proceeds.

You can have each replica create its own dump file in the following manner:

```
variable rep world 0 1 2 3 4 5 6 7
dump 1 all atom 1000 dump.temper.${rep}
```

Note

Each replica's dump file will contain a continuous trajectory for its atoms where the temperature varies over time as swaps take place involving that replica. If you want a series of dump files, each with snapshots (from all replicas) that are all at a single temperature, then you will need to post-process the dump files using the information from the log.lammps file. E.g. you could produce one dump file with snapshots at 300K (from all replicas), another with snapshots at 310K, etc. Note that these new dump files will not contain "continuous trajectories" for individual atoms, because two successive snapshots (in time) may be from different replicas. The reorder_remd_traj python script can do the reordering for you (and additionally also calculated configurational log-weights of trajectory snapshots in the canonical ensemble). The script can be found in the tools/replica directory while instructions on how to use it is available in doc/Tools (in brief) and as a README file in tools/replica (in detail).

The last argument *index* in the temper command is optional and is used when restarting a tempering run from a set of restart files (one for each replica) which had previously swapped to new temperatures. The *index* value (from 0 to N-1, where N is the # of replicas) identifies which temperature the replica was simulating on the timestep the restart files were written. Obviously, this argument must be a variable so that each partition has the correct value. Set the variable to the *N* values listed in the log file for the previous run for the replica temperatures at that timestep. For example if the log file listed the following for a simulation with 5 replicas:

```
500000 2 4 0 1 3
```

then a setting of

```
variable w world 2 4 0 1 3
```

would be used to restart the run with a tempering command like the example above with $w as the last argument.

1.101.4 Restrictions

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

1.101.5 Related commands

variable, *prd*, *neb*

1.101.6 Default

none

1.102 temper/grem command

1.102.1 Syntax

```
temper/grem N M lambda fix-ID thermostat-ID seed1 seed2 index
```

- N = total # of timesteps to run
- M = attempt a tempering swap every this many steps
- lambda = initial lambda for this ensemble
- fix-ID = ID of *fix grem*
- thermostat-ID = ID of the thermostat that controls kinetic temperature
- seed1 = random # seed used to decide on adjacent temperature to partner with
- seed2 = random # seed for Boltzmann factor in Metropolis swap
- index = which temperature (0 to N-1) I am simulating (optional)

1.102.2 Examples

```
temper/grem 100000 1000 ${lambda} fxgREM fxnvt 0 58728
temper/grem 40000 100 ${lambda} fxgREM fxnpt 0 32285 ${walkers}
```

1.102.3 Description

Run a parallel tempering or replica exchange simulation in LAMMPS partition mode using multiple generalized replicas (ensembles) of a system defined by *fix grem*, which stands for the generalized replica exchange method (gREM) originally developed by *(Kim)*. It uses non-Boltzmann ensembles to sample over first order phase transitions. The is done by defining replicas with an enthalpy dependent effective temperature

Two or more replicas must be used. See the *temper* command for an explanation of how to run replicas on multiple partitions of one or more processors.

This command is a modification of the *temper* command and has the same dependencies, restraints, and input variables which are discussed there in greater detail.

Instead of temperature, this command performs replica exchanges in lambda as per the generalized ensemble enforced by *fix grem*. The desired lambda is specified by *lambda*, which is typically a variable previously set in the input script, so that each partition is assigned a different temperature. See the *variable* command for more details. For example:

```
variable lambda world 400 420 440 460
fix fxnvt all nvt temp 300.0 300.0 100.0
fix fxgREM all grem ${lambda} -0.05 -50000 fxnvt
temper/grem 100000 100 ${lambda} fxgREM fxnvt 3847 58382
```

would define 4 lambdas with constant kinetic temperature but unique generalized temperature, and assign one of them to *fix grem* used by each replica, and to the grem command.

As the gREM simulation runs for *N* timesteps, a swap between adjacent ensembles will be attempted every *M* timesteps. If *seed1* is 0, then the swap attempts will alternate between odd and even pairings. If *seed1* is non-zero then it is used as a seed in a random number generator to randomly choose an odd or even pairing each time. Each attempted swap of temperatures is either accepted or rejected based on a Metropolis criterion, derived for gREM by *(Kim)*, which uses *seed2* in the random number generator.

File management works identical to the *temper* command. Dump files created by this fix contain continuous trajectories and require post-processing to obtain per-replica information.

The last argument *index* in the grem command is optional and is used when restarting a run from a set of restart files (one for each replica) which had previously swapped to new lambda. This is done using a variable. For example if the log file listed the following for a simulation with 5 replicas:

```
500000 2 4 0 1 3
```

then a setting of

```
variable walkers world 2 4 0 1 3
```

would be used to restart the run with a grem command like the example above with ${walkers} as the last argument. This functionality is identical to *temper*.

1.102.4 Restrictions

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

This command must be used with *fix grem*.

1.102.5 Related commands

fix grem, *temper*, *variable*

1.102.6 Default

none

(Kim) Kim, Keyes, Straub, J Chem Phys, 132, 224107 (2010).

1.103 temper/npt command

1.103.1 Syntax

```
temper/npt  N M temp fix-ID seed1 seed2 pressure index
```

- N = total # of timesteps to run
- M = attempt a tempering swap every this many steps
- temp = initial temperature for this ensemble
- fix-ID = ID of the fix that will control temperature and pressure during the run
- seed1 = random # seed used to decide on adjacent temperature to partner with
- seed2 = random # seed for Boltzmann factor in Metropolis swap
- pressure = setpoint pressure for the ensemble
- index = which temperature (0 to N-1) I am simulating (optional)

1.103.2 Examples

```
temper/npt 100000 100 $t nptfix 0 58728 1
temper/npt 2500000 1000 300 nptfix  0 32285 $p
temper/npt 5000000 2000 $t nptfix 0 12523 1 $w
```

1.103.3 Description

Run a parallel tempering or replica exchange simulation using multiple replicas (ensembles) of a system in the isothermal-isobaric (NPT) ensemble. The command temper/npt works like *temper* but requires running replicas in the NPT ensemble instead of the canonical (NVT) ensemble and allows for pressure to be set in the ensembles. These multiple ensembles can run in parallel at different temperatures or different pressures. The acceptance criteria for temper/npt is specific to the NPT ensemble and can be found in references *(Okabe)* and *(Mori)*.

Apart from the difference in acceptance criteria and the specification of pressure, this command works much like the *temper* command. See the documentation on *temper* for information on how the parallel tempering is handled in general.

1.103.4 Restrictions

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

This command should be used with a fix that maintains the isothermal-isobaric (NPT) ensemble.

1.103.5 Related commands

temper, *variable*, *fix_npt*

1.103.6 Default

none

(Okabe) T. Okabe, M. Kawata, Y. Okamoto, M. Masuhiro, Chem. Phys. Lett., 335, 435-439 (2001).

(Mori) Y. Mori, Y. Okamoto, J. Phys. Soc. Jpn., 7, 074003 (2010).

1.104 thermo command

1.104.1 Syntax

```
thermo N
```

- N = output thermodynamics every N timesteps
- N can be a variable (see below)

1.104.2 Examples

```
thermo 100
```

1.104.3 Description

Compute and print thermodynamic info (e.g. temperature, energy, pressure) on timesteps that are a multiple of N and at the beginning and end of a simulation. A value of 0 will only print thermodynamics at the beginning and end.

The content and format of what is printed is controlled by the *thermo_style* and *thermo_modify* commands.

Instead of a numeric value, N can be specified as an *equal-style variable*, which should be specified as v_name, where name is the variable name. In this case, the variable is evaluated at the beginning of a run to determine the next timestep at which thermodynamic info will be written out. On that timestep, the variable will be evaluated again to determine the next timestep, etc. Thus the variable should return timestep values. See the stagger() and logfreq() and stride() math functions for *equal-style variables*, as examples of useful functions to use in this context. Other similar math functions could easily be added as options for *equal-style variables*.

For example, the following commands will output thermodynamic info at timesteps 0, 10, 20, 30, 100, 200, 300, 1000, 2000, *etc*:

```
variable    s equal logfreq(10,3,10)
thermo      v_s
```

1.104.4 Restrictions

none

1.104.5 Related commands

thermo_style, *thermo_modify*

1.104.6 Default

```
thermo 0
```

1.105 thermo_modify command

1.105.1 Syntax

```
thermo_modify keyword value ...
```

- one or more keyword/value pairs may be listed
- keyword = *lost* or *lost/bond* or *warn* or *norm* or *flush* or *line* or *colname* or *format* or *temp* or *press* or *triclinic/general*

lost value = error or warn or ignore
lost/bond value = error or warn or ignore
warn value = ignore or reset or default or a number
norm value = yes or no
flush value = yes or no
line value = one or multi or yaml
colname values = ID string, or default
  string = new column header name
  ID = integer from 1 to N, or integer from -1 to -N, where N = # of quantities being output
    or a thermo keyword or reference to compute, fix, property or variable.
format values = line string, int string, float string, ID string, or none
  string = C-style format string
  ID = integer from 1 to N, or integer from -1 to -N, where N = # of quantities being output
    or an integer range such as 2*6 (negative values are not allowed)
    or a thermo keyword or reference to compute, fix, property or variable.
temp value = compute ID that calculates a temperature
press value = compute ID that calculates a pressure
triclinic/general arg = yes or no

1.105.2 Examples

```
thermo_modify lost ignore flush yes
thermo_modify temp myTemp format 3 %15.8g
thermo_modify temp myTemp format line "%ld %g %g %15.8g"
thermo_modify line multi format float %g
thermo_modify line yaml format none
thermo_modify colname 1 Timestep colname -2 Pressure colname f_1[1] AvgDensity
```

1.105.3 Description

Set options for how thermodynamic information is computed and printed by LAMMPS.

Note

These options apply to the *currently defined* thermo style. When you specify a *thermo_style* command, all thermodynamic settings are restored to their default values, including those previously reset by a thermo_modify command. Thus if your input script specifies a thermo_style command, you should use the thermo_modify command **after** it.

The *lost* keyword determines whether LAMMPS checks for lost atoms each time it computes thermodynamics and what it does if atoms are lost. An atom can be "lost" if it moves across a non-periodic simulation box *boundary* or if it moves more than a box length outside the simulation domain (or more than a processor subdomain length) before reneighboring occurs. The latter case is typically due to bad dynamics (e.g., too large a time step and/or huge forces and velocities). 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 simulation will exit with an error and continue with a warning. A warning will only be issued once, the first time an atom is lost. This can be a useful debugging option.

The *lost/bond* keyword determines whether LAMMPS throws an error or not if an atom in a bonded interaction (bond, angle, etc) cannot be found when it creates bonded neighbor lists. By default this is a fatal error. However in some scenarios it may be desirable to only issue a warning or ignore it and skip the computation of the missing bond, angle, etc. An example would be when gas molecules in a vapor are drifting out of the box through a fixed boundary condition (see the *boundary* command). In this case one atom may be deleted before the rest of the molecule is, on a later timestep.

The *warn* keyword allows you to control whether LAMMPS will print warning messages and how many of them. Most warning messages are only printed by MPI rank 0. They are usually pointing out important issues that should be investigated, but LAMMPS cannot determine for certain whether they are an indication of an error.

Some warning messages are printed during a run (or immediately before) each time a specific MPI rank encounters the issue (e.g., bonds that are stretched too far or dihedrals in extreme configurations). These number of these can quickly blow up the size of the log file and screen output. Thus, a limit of 100 warning messages is applied by default. The warning count is applied to the entire input unless reset with a thermo_modify warn reset command. If there are more warnings than the limit, LAMMPS will print one final warning that it will not print any additional warning messages.

Note

The warning limit is enforced on either the per-processor count or the total count across all processors. For efficiency reasons, however, the total count is only updated at steps with thermodynamic output. Thus when running on a large number of processors in parallel, the total number of warnings printed can be significantly larger than the given limit.

Any number after the keyword *warn* will change the warning limit accordingly. With the value *ignore* all warnings will be suppressed, with the value *always* no limit will be applied and warnings will always be printed, with the value *reset* the internal warning counter will be reset to zero, and with the value *default*, the counter is reset and the limit set to 100. An example usage of either *reset* or *default* would be to re-enable warnings that were disabled or have reached the limit during equilibration, where the warnings would be acceptable while the system is still adjusting, but then change to all warnings for the production run, where they would indicate problems that would require a closer look at what is causing them.

The *norm* keyword determines whether various thermodynamic output values are normalized by the number of atoms or not, depending on whether it is set to *yes* or *no*. Different unit styles have different defaults for this setting (see below). Even if *norm* is set to *yes*, a value is only normalized if it is an "extensive" quantity, meaning that it scales with the number of atoms in the system. For the thermo keywords described by the page for the *thermo_style* command, all energy-related keywords are extensive, such as *pe* or *ebond* or *enthalpy*. Other keywords such as *temp* or *press* are "intensive" meaning their value is independent (in a statistical sense) of the number of atoms in the system and thus are never normalized. For thermodynamic output values extracted from fixes and computes in a *thermo_style custom* command, the page for the individual *fix* or *compute* lists whether the value is "extensive" or "intensive" and thus whether it is normalized. Thermodynamic output values calculated by a variable formula are assumed to be "intensive" and thus are never normalized. You can always include a divide by the number of atoms in the variable formula if this is not the case.

The *flush* keyword invokes a flush operation after thermodynamic info is written to the screen and log file. This ensures the output is updated and not buffered (by the application) even if LAMMPS halts before the simulation completes. Please note that this does not affect buffering by the OS or devices, so you may still lose data in case the simulation stops due to a hardware failure.

The *line* keyword determines whether thermodynamics will be output as a series of numeric values on one line ("one"), in a multi-line format with 3 quantities with text strings per line and a dashed-line header containing the timestep and CPU time ("multi"), or in a YAML format block ("yaml"). This modify option overrides the *one*, *multi*, or *yaml* thermo_style settings.

Added in version 4May2022.

The *colname* keyword can be used to change the default header keyword for a column or field of thermodynamic output. The setting for *ID string* replaces the default text with the provided string. *ID* can be a positive integer when it represents the column number counting from the left, a negative integer when it represents the column number from the right (i.e., -1 is the last column/keyword), or a thermo keyword (or compute, fix, property, or variable reference) and then it replaces the string for that specific thermo keyword.

The *colname* keyword can be used multiple times. If multiple *colname* settings refer to the same keyword, the last setting has precedence. A setting of *default* clears all previous settings, reverting all values to their default values.

The *format* keyword can be used to change the default numeric format of any of quantities the *thermo_style* command outputs. All the specified format strings are C-style formats (i.e., as used by the C/C++ printf() command). The *line* keyword takes a single argument which is the format string for the entire line of thermo output, with *N* fields, which you must enclose in quotes if it is more than one field. The *int* and *float* keywords take a single format argument and are applied to all integer or floating-point quantities output. The setting for *ID string* also takes a single format argument that is used for the indexed value in each line. The interpretation is the same as for *colname* (i.e., a positive integer is the n-th value corresponding to the n-th thermo keyword, a negative integer is counting backwards, and a string matches the entry with the thermo keyword). For example, the fifth column is output in high precision for "format 5 %20.15g", and the pair energy for "format epair %20.15g". The *ID* field can be a range, such as "3*6", "*", "2*", or "*3"; in such cases, all fields in the range (inclusive) are set to the specified format string. Ranges containing negative numbers are not supported.

The *format* keyword can be used multiple times. The precedence is that for each value in a line of output, the *ID* format (if specified) is used, else the *int* or *float* setting (if specified) is used, else the *line* setting (if specified) for that value is used, else the default setting is used. A setting of *none* clears all previous settings, reverting all values to their default format.

Note

The thermo output values *step* and *atoms* are stored internally as 8-byte signed integers, rather than the usual 4-byte signed integers. When specifying the *format int* option you can use a "%d"-style format identifier in the format string and LAMMPS will convert this to the corresponding 8-byte form when it is applied to those keywords. However, when specifying the *line* option or *format ID string* option for *step* and *natoms*, you should specify a format string appropriate for an 8-byte signed integer (i.e., one with "%ld" or "%lld", depending on the platform).

The *temp* keyword is used to determine how thermodynamic temperature is calculated, which is used by all thermo quantities that require a temperature ("temp", "press", "ke", "etotal", "enthalpy", "pxx", etc). The specified compute ID must have been previously defined by the user via the *compute* command and it must be a style of compute that calculates a temperature. As described in the *thermo_style* command, thermo output uses a default compute for temperature with ID = *thermo_temp*. This option allows the user to override the default.

The *press* keyword is used to determine how thermodynamic pressure is calculated, which is used by all thermo quantities that require a pressure ("press", "enthalpy", "pxx", etc). The specified compute ID must have been previously defined by the user via the *compute* command and it must be a style of compute that calculates a pressure. As described in the *thermo_style* command, thermo output uses a default compute for pressure with ID = *thermo_press*. This option allows the user to override the default.

Note

If both the *temp* and *press* keywords are used in a single thermo_modify command (or in two separate commands), then the order in which the keywords are specified is important. Note that a *pressure compute* defines its own temperature compute as an argument when it is specified. The *temp* keyword will override this (for the pressure compute being used by thermodynamics), but only if the *temp* keyword comes after the *press* keyword. If the *temp* keyword comes before the *press* keyword, then the new pressure compute specified by the *press* keyword will be unaffected by the *temp* setting.

The *triclinic/general* keyword can only be used with a value of *yes* if the simulation box was created as a general triclinic box. See the *Howto_triclinic* doc page for a detailed explanation of orthogonal, restricted triclinic, and general triclinic simulation boxes.

If this keyword is *yes*, the output of the simulation box edge vectors and the pressure tensor components for the sys-

tem are affected. These are specified by the *avec,bvec,cvec* and *pxx,pyy,pzz,pxy,pxz,pyz* keywords of the *thermo_style* command. See the *thermo_style* doc page for details.

1.105.4 Restrictions

none

1.105.5 Related commands

thermo, *thermo_style*

1.105.6 Default

The option defaults are lost = error, warn = 100, norm = yes for unit style of *lj*, norm = no for unit style of *real* and *metal*, flush = no, temp/press = compute IDs defined by thermo_style, and triclinic/general = no.

The defaults for the line and format options depend on the thermo style. For styles "one" and "custom", the line and format defaults are "one", "%10d", and "%14.8g". For style "multi", the line and format defaults are "multi", "%14d", and "%14.4f". For style "yaml", the line and format defaults are "%d" and "%.15g".

1.106 thermo_style command

1.106.1 Syntax

```
thermo_style style args
```

- style = *one* or *multi* or *yaml* or *custom*
- args = list of arguments for a particular style

```
one args = none
multi args = none
yaml args = none
custom args = list of keywords
  possible keywords = step, elapsed, elaplong, dt, time,
                      cpu, tpcpu, spcpu, cpuremain, part, timeremain,
                      atoms, temp, press, pe, ke, etotal,
                      evdwl, ecoul, epair, ebond, eangle, edihed, eimp,
                      emol, elong, etail,
                      enthalpy, ecouple, econserve,
                      vol, density,
                      xlo, xhi, ylo, yhi, zlo, zhi,
                      xy, xz, yz,
                      avecx, avecy, avecz,
                      bvecx, bvecy, bvecz,
                      cvecx, cvecy, cvecz,
                      lx, ly, lz,
                      xlat, ylat, zlat,
                      cella, cellb, cellc, cellalpha, cellbeta, cellgamma,
                      pxx, pyy, pzz, pxy, pxz, pyz,
                      bonds, angles, dihedrals, impropers,
```

fmax, fnorm, nbuild, ndanger,
c_ID, c_ID[I], c_ID[I][J],
f_ID, f_ID[I], f_ID[I][J],
v_name, v_name[I]
step = timestep
elapsed = timesteps since start of this run
elaplong = timesteps since start of initial run in a series of runs
dt = timestep size
time = simulation time
cpu = elapsed CPU time in seconds since start of this run
tpcpu = time per CPU second
spcpu = timesteps per CPU second
cpuremain = estimated CPU time remaining in run
part = which partition (0 to Npartition-1) this is
timeremain = remaining time in seconds on timer timeout.
atoms = # of atoms
temp = temperature
press = pressure
pe = total potential energy
ke = kinetic energy
etotal = total energy (pe + ke)
evdwl = van der Waals pairwise energy (includes etail)
ecoul = Coulombic pairwise energy
epair = pairwise energy (evdwl + ecoul + elong)
ebond = bond energy
eangle = angle energy
edihed = dihedral energy
eimp = improper energy
emol = molecular energy (ebond + eangle + edihed + eimp)
elong = long-range kspace energy
etail = van der Waals energy long-range tail correction
enthalpy = enthalpy (etotal + press*vol)
ecouple = cumulative energy change due to thermo/baro statting fixes
econserve = pe + ke + ecouple = etotal + ecouple
vol = volume
density = mass density of system
xlo,xhi,ylo,yhi,zlo,zhi = box boundaries
xy,xz,yz = box tilt for restricted triclinic (non-orthogonal) simulation boxes
avecx,avecy,avecz = components of edge vector A of the simulation box
bvecx,bvecy,bvecz = components of edge vector B of the simulation box
cvecx,cvecy,cvecz = components of edge vector C of the simulation box
lx,ly,lz = box lengths in x,y,z
xlat,ylat,zlat = lattice spacings as calculated by lattice command
cella,cellb,cellc = periodic cell lattice constants a,b,c
cellalpha, cellbeta, cellgamma = periodic cell angles alpha,beta,gamma
pxx,pyy,pzz,pxy,pxz,pyz = 6 components of pressure tensor
bonds,angles,dihedrals,impropers = # of these interactions defined
fmax = max component of force on any atom in any dimension
fnorm = length of force vector for all atoms
nbuild = # of neighbor list builds
ndanger = # of dangerous neighbor list builds
c_ID = global scalar value calculated by a compute with ID
c_ID[I] = Ith component of global vector calculated by a compute with ID, I can include␣
↪wildcard (see below)

c_ID[I][J] = I,J component of global array calculated by a compute with ID
f_ID = global scalar value calculated by a fix with ID
f_ID[I] = Ith component of global vector calculated by a fix with ID, I can include wildcard (see below)
f_ID[I][J] = I,J component of global array calculated by a fix with ID
v_name = value calculated by an equal-style variable with name
v_name[I] = value calculated by a vector-style variable with name, I can include wildcard (see below)

1.106.2 Examples

```
thermo_style multi
thermo_style yaml
thermo_style one
thermo_style custom step temp pe etotal press vol
thermo_style custom step temp etotal c_myTemp v_abc
thermo_style custom step temp etotal c_myTemp[*] v_abc
```

1.106.3 Description

Set the style and content for printing thermodynamic data to the screen and log files. The units for each column of output corresponding to the list of keywords is determined by the *units* command for the simulation. E.g. energies will be in energy units, temperature in temperature units, pressure in pressure units.

Style *one* prints a single line of thermodynamic info that is the equivalent of "thermo_style custom step temp epair emol etotal press". The line contains only numeric values.

Style *multi* prints a multiple-line listing of thermodynamic info that is the equivalent of "thermo_style custom etotal ke temp pe ebond eangle edihed eimp evdwl ecoul elong press". The listing contains numeric values and a string ID for each quantity.

Added in version 24Mar2022.

Style *yaml* is similar to style *one* but prints the output in YAML format which can be easily read by a variety of script languages and data handling packages. Since LAMMPS may print other output before, after, or in between thermodynamic output, the YAML format content needs to be separated from the rest. All YAML format thermodynamic output can be matched with a regular expression and can thus be extracted with commands like egrep as follows:

```
egrep '^(keywords:|data:$|---$|\.\.\.$| - \[)' log.lammps > log.yaml
```

Information about processing such YAML files is in the *structured data output howto*.

Style *custom* is the most general setting and allows you to specify which of the keywords listed above you want printed on each thermodynamic timestep. Note that the keywords c_ID, f_ID, v_name are references to *computes*, *fixes*, and equal-style *variables* that have been defined elsewhere in the input script or can even be new styles which users have added to LAMMPS. See the *Modify* page for details on the latter. Thus the *custom* style provides a flexible means of outputting essentially any desired quantity as a simulation proceeds.

All styles except *custom* have *vol* appended to their list of outputs if the simulation box volume changes during the simulation.

The values printed by the various keywords are instantaneous values, calculated on the current timestep. Time-averaged quantities, which include values from previous timesteps, can be output by using the f_ID keyword and accessing a fix that does time-averaging such as the *fix ave/time* command.

Options invoked by the *thermo_modify* command can be used to set the one- or multi-line format of the print-out, the normalization of thermodynamic output (total values versus per-atom values for extensive quantities (ones which scale with the number of atoms in the system), and the numeric precision of each printed value.

Note

When you use a "thermo_style" command, all thermodynamic settings are restored to their default values, including those previously set by a *thermo_modify* command. Thus if your input script specifies a thermo_style command, you should use the thermo_modify command after it.

Several of the thermodynamic quantities require a temperature to be computed: "temp", "press", "ke", "etotal", "enthalpy", "pxx", etc. By default this is done by using a *temperature* compute which is created when LAMMPS starts up, as if this command had been issued:

```
compute thermo_temp all temp
```

See the *compute temp* command for details. Note that the ID of this compute is *thermo_temp* and the group is *all*. You can change the attributes of this temperature (e.g. its degrees-of-freedom) via the *compute_modify* command. Alternatively, you can directly assign a new compute (that calculates temperature) which you have defined, to be used for calculating any thermodynamic quantity that requires a temperature. This is done via the *thermo_modify* command.

Several of the thermodynamic quantities require a pressure to be computed: "press", "enthalpy", "pxx", etc. By default this is done by using a *pressure* compute which is created when LAMMPS starts up, as if this command had been issued:

```
compute thermo_press all pressure thermo_temp
```

See the *compute pressure* command for details. Note that the ID of this compute is *thermo_press* and the group is *all*. You can change the attributes of this pressure via the *compute_modify* command. Alternatively, you can directly assign a new compute (that calculates pressure) which you have defined, to be used for calculating any thermodynamic quantity that requires a pressure. This is done via the *thermo_modify* command.

Several of the thermodynamic quantities require a potential energy to be computed: "pe", "etotal", "ebond", etc. This is done by using a *pe* compute which is created when LAMMPS starts up, as if this command had been issued:

```
compute thermo_pe all pe
```

See the *compute pe* command for details. Note that the ID of this compute is *thermo_pe* and the group is *all*. You can change the attributes of this potential energy via the *compute_modify* command.

The kinetic energy of the system *ke* is inferred from the temperature of the system with $\frac{1}{2}k_BT$ of energy for each degree of freedom. Thus, using different *compute commands* for calculating temperature, via the *thermo_modify temp* command, may yield different kinetic energies, since different computes that calculate temperature can subtract out different non-thermal components of velocity and/or include different degrees of freedom (translational, rotational, etc).

The potential energy of the system *pe* will include contributions from fixes if the *fix_modify energy yes* option is set for a fix that calculates such a contribution. For example, the *fix wall/lj93* fix calculates the energy of atoms interacting with the wall. See the doc pages for "individual fixes" to see which ones contribute and whether their default *fix_modify energy* setting is *yes* or *no*.

A long-range tail correction *etail* for the van der Waals pairwise energy will be non-zero only if the *pair_modify tail* option is turned on. The *etail* contribution is included in *evdwl*, *epair*, *pe*, and *etotal*, and the corresponding tail correction to the pressure is included in *press* and *pxx*, *pyy*, etc.

Here is more information on other keywords whose meaning may not be clear.

The *step*, *elapsed*, and *elaplong* keywords refer to timestep count. *Step* is the current timestep, or iteration count when a *minimization* is being performed. *Elapsed* is the number of timesteps elapsed since the beginning of this run. *Elaplong* is the number of timesteps elapsed since the beginning of an initial run in a series of runs. See the *start* and *stop* keywords for the *run* for info on how to invoke a series of runs that keep track of an initial starting time. If these keywords are not used, then *elapsed* and *elaplong* are the same value.

The *dt* keyword is the current timestep size in time *units*. The *time* keyword is the current elapsed simulation time, also in time *units*, which is simply (step*dt) if the timestep size has not changed and the timestep has not been reset. If the timestep has changed (e.g. via *fix dt/reset*) or the timestep has been reset (e.g. via the "reset_timestep" command), then the simulation time is effectively a cumulative value up to the current point.

The *cpu* keyword is elapsed CPU seconds since the beginning of this run. The *tpcpu* and *spcpu* keywords are measures of how fast your simulation is currently running. The *tpcpu* keyword is simulation time per CPU second, where simulation time is in time *units*. E.g. for metal units, the *tpcpu* value would be picoseconds per CPU second. The *spcpu* keyword is the number of timesteps per CPU second. Both quantities are on-the-fly metrics, measured relative to the last time they were invoked. Thus if you are printing out thermodynamic output every 100 timesteps, the two keywords will continually output the time and timestep rate for the last 100 steps. The *tpcpu* keyword does not attempt to track any changes in timestep size, e.g. due to using the *fix dt/reset* command.

The *cpuremain* keyword estimates the CPU time remaining in the current run, based on the time elapsed thus far. It will only be a good estimate if the CPU time/timestep for the rest of the run is similar to the preceding timesteps. On the initial timestep the value will be 0.0 since there is no history to estimate from. For a minimization run performed by the "minimize" command, the estimate is based on the *maxiter* parameter, assuming the minimization will proceed for the maximum number of allowed iterations.

The *part* keyword is useful for multi-replica or multi-partition simulations to indicate which partition this output and this file corresponds to, or for use in a *variable* to append to a filename for output specific to this partition. See discussion of the *-partition command-line switch* for details on running in multi-partition mode.

The *timeremain* keyword is the seconds remaining when a timeout has been configured via the *timer timeout* command. If the timeout timer is inactive, the value of this keyword is 0.0 and if the timer is expired, it is negative. This allows for example to exit loops cleanly, if the timeout is expired with:

```
if "$(timeremain) < 0.0" then "quit 0"
```

The *ecouple* keyword is cumulative energy change in the system due to any thermostatting or barostatting fixes that are being used. A positive value means that energy has been subtracted from the system (added to the coupling reservoir). See the *econserve* keyword for an explanation of why this sign choice makes sense.

The *econserve* keyword is the sum of the potential and kinetic energy of the system as well as the energy that has been transferred by thermostatting or barostatting to their coupling reservoirs – that is, *econserve* = *pe* + *ke* + *ecouple*. Ideally, for a simulation in the NVT, NPH, or NPT ensembles, the *econserve* quantity should remain constant over time even though *etotal* may change.

In LAMMPS, the simulation box can be defined as orthogonal or triclinic (non-orthogonal). See the *Howto_triclinic* doc page for a detailed explanation of orthogonal, restricted triclinic, and general triclinic simulation boxes and how LAMMPS rotates a general triclinic box to be restricted triclinic internally.

The *lx*, *ly*, *lz* keywords are the extent of the simulation box in each dimension. The *xlo*, *xhi*, *ylo*, *yhi*, *zlo*, *zhi* keywords are the lower and upper bounds of the simulation box in each dimension. I.e. *lx* = *xhi* - *xlo*). These 9 values are the same for all 3 kinds of boxes. I.e. for a restricted triclinic box, they are the values as if the box were not tilted. For a general triclinic box, they are the values after it is internally rotated to be a restricted triclinic box.

The *xy*, *xz*, *yz* are the current tilt factors for a triclinic box. They are the same for restricted and general triclinic boxes.

The *avecx*, *avecy*, *avecz*, *bvecx*, *bvecy*, *bvecz*, *cvecx*, *cvecy*, *cvecz* are the components of the 3 edge vectors of the current general simulation box. If it is an orthogonal box the vectors are along the x, y, z coordinate axes. If it is a restricted triclinic box, the **A** vector is along the x axis, the **B** vector is in the xy plane with a +y coordinate, and the **C** vector has a +z coordinate, as explained on the *Howto_triclinic* doc page. If the *thermo_modify triclinic/general* option is set then they are the **A**, **B**, **C** vector which define the general triclinic box.

The *cella*, *cellb*, *cellc*, *cellalpha*, *cellbeta*, *cellgamma* keywords correspond to the usual crystallographic quantities that define the periodic simulation box of a crystalline system. See the *Howto triclinic* page for a precise definition of these quantities in terms of the LAMMPS representation of a restricted triclinic simulation box via *lx*, *ly*, *lz*, *yz*, *xz*, *xy*.

The *pxx,pyy,pzz,pxy,pxz,pyz* keywords are the 6 components of the symmetric pressure tensor for the system. See the *compute pressure* command doc page for details of how it is calculated.

If the *thermo_modify triclinic/general* option is set then the 6 components will be output as values consistent with the orientation of the general triclinic box relative to the standard xyz coordinate axes. If this keyword is not used, the values will be consistent with the orientation of the restricted triclinic box (which aligns with the xyz coordinate axes). As explained on the *Howto_triclinic* doc page, even if the simulation box is created as a general triclinic box, internally LAMMPS uses a restricted triclinic box.

Note that because the pressure tensor components are computed using force vectors and atom coordinates, both of which are rotated in the general versus restricted triclinic representation, the values will typically be different for the two cases.

The *fmax* and *fnorm* keywords are useful for monitoring the progress of an *energy minimization*. The *fmax* keyword calculates the maximum force in any dimension on any atom in the system, or the infinity-norm of the force vector for the system. The *fnorm* keyword calculates the 2-norm or length of the force vector.

The *nbuild* and *ndanger* keywords are useful for monitoring neighbor list builds during a run. Note that both these values are also printed with the end-of-run statistics. The *nbuild* keyword is the number of re-builds during the current run. The *ndanger* keyword is the number of re-builds that LAMMPS considered potentially "dangerous". If atom movement triggered neighbor list rebuilding (see the *neigh_modify* command), then dangerous reneighborings are those that were triggered on the first timestep atom movement was checked for. If this count is non-zero you may wish to reduce the delay factor to ensure no force interactions are missed by atoms moving beyond the neighbor skin distance before a rebuild takes place.

For output values from a compute or fix or variable, the bracketed index I used to index a vector, as in *c_ID[I]* or *f_ID[I]* or *v_name[I]*, can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "n*" or "m*n". If N = the size of the vector, then an asterisk with no numeric values means all indices from 1 to N. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from n to N (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual elements of the vector had been listed one by one. E.g. these 2 thermo_style commands are equivalent, since the *compute temp* command creates a global vector with 6 values.

```
compute myTemp all temp
thermo_style custom step temp etotal c_myTemp[*]
thermo_style custom step temp etotal &
             c_myTemp[1] c_myTemp[2] c_myTemp[3] &
             c_myTemp[4] c_myTemp[5] c_myTemp[6]
```

Note

For a vector-style variable, only the wildcard forms "*n" or "m*n" are allowed. You must specify the upper bound, because vector-style variable lengths are not determined until the variable is evaluated. If n is specified larger than the vector length turns out to be, zeroes are output for missing vector values.

The *c_ID* and *c_ID[I]* and *c_ID[I][J]* keywords allow global values calculated by a compute to be output. As discussed on the *compute* doc page, computes can calculate global, per-atom, local, and per-grid values. Only global values can be referenced by this command. However, per-atom compute values for an individual atom can be referenced in a *equal-style variable* and the variable referenced by thermo_style custom, as discussed below. See the discussion above for how the I in *c_ID[I]* can be specified with a wildcard asterisk to effectively specify multiple values from a global compute vector.

The ID in the keyword should be replaced by the actual ID of a compute that has been defined elsewhere in the input script. See the *compute* command for details. If the compute calculates a global scalar, vector, or array, then the keyword formats with 0, 1, or 2 brackets will reference a scalar value from the compute.

Note that some computes calculate "intensive" global quantities like temperature; others calculate "extensive" global quantities like kinetic energy that are summed over all atoms in the compute group. Intensive quantities are printed directly without normalization by thermo_style custom. Extensive quantities may be normalized by the total number of atoms in the simulation (NOT the number of atoms in the compute group) when output, depending on the *thermo_modify norm* option being used.

The *f_ID* and *f_ID[I]* and *f_ID[I][J]* keywords allow global values calculated by a fix to be output. As discussed on the *fix* doc page, fixes can calculate global, per-atom, local, and per-grid values. Only global values can be referenced by this command. However, per-atom fix values can be referenced for an individual atom in a *equal-style variable* and the variable referenced by thermo_style custom, as discussed below. See the discussion above for how the I in *f_ID[I]* can be specified with a wildcard asterisk to effectively specify multiple values from a global fix vector.

The ID in the keyword should be replaced by the actual ID of a fix that has been defined elsewhere in the input script. See the *fix* command for details. If the fix calculates a global scalar, vector, or array, then the keyword formats with 0, 1, or 2 brackets will reference a scalar value from the fix.

Note that some fixes calculate "intensive" global quantities like timestep size; others calculate "extensive" global quantities like energy that are summed over all atoms in the fix group. Intensive quantities are printed directly without normalization by thermo_style custom. Extensive quantities may be normalized by the total number of atoms in the simulation (NOT the number of atoms in the fix group) when output, depending on the *thermo_modify norm* option being used.

The *v_name* keyword allow the current value of a variable to be output. The name in the keyword should be replaced by the variable name that has been defined elsewhere in the input script. Only equal-style and vector-style variables can be referenced; the latter requires a bracketed term to specify the Ith element of the vector calculated by the variable. However, an equal-style variable can use an atom-style variable in its formula indexed by the ID of an individual atom. This is a way to output a specific atom's per-atom coordinates or other per-atom properties in thermo output. See the *variable* command for details. Note that variables of style *equal* and *vector* and *atom* define a formula which can reference per-atom properties or thermodynamic keywords, or they can invoke other computes, fixes, or variables when evaluated, so this is a very general means of creating thermodynamic output.

Note that equal-style and vector-style variables are assumed to produce "intensive" global quantities, which are thus printed as-is, without normalization by thermo_style custom. You can include a division by "natoms" in the variable formula if this is not the case.

1.106.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

1.106.5 Related commands

thermo, *thermo_modify*, *fix_modify*, *compute temp*, *compute pressure*

1.106.6 Default

```
thermo_style one
```

1.107 third_order command

Accelerator Variant: third_order/kk

1.107.1 Syntax

```
third_order group-ID style delta args keyword value ...
```

- group-ID = ID of group of atoms to displace
- style = *regular* or *eskm*
- delta = finite different displacement length (distance units)
- one or more keyword/arg pairs may be appended

  keyword = file or binary
    file name = name of output file for the third order tensor
    binary arg = yes or no or gzip

1.107.2 Examples

```
third_order 1 regular 0.000001
third_order 1 eskm 0.000001
third_order 3 regular 0.00004 file third_order.dat
third_order 5 eskm 0.00000001 file third_order.dat binary yes
```

1.107.3 Description

Calculate the third order force constant tensor by finite difference of the selected group,

$$\Phi_{ijk}^{\alpha\beta\gamma} = \frac{\partial^3 U}{\partial x_{i,\alpha} \partial x_{j,\beta} \partial x_{k,\gamma}}$$

where Phi is the third order force constant tensor.

The output of the command is the tensor, three elements at a time. The three elements correspond to the three gamma elements for a specific i/alpha/j/beta/k. The initial five numbers are i, alpha, j, beta, and k respectively.

If the style eskm is selected, the tensor will be using energy units of 10 J/mol. These units conform to eskm style from the dynamical_matrix command, which will simplify operations using dynamical matrices with third order tensors.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

1.107.4 Restrictions

The command collects a 9 times the number of atoms in the group on every single MPI rank, so the memory requirements can be very significant for large systems.

This command is part of the PHONON package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

1.107.5 Related commands

fix phonon dynamical_matrix

1.107.6 Default

The default settings are file = “third_order.dat”, binary = no

1.108 timer command

1.108.1 Syntax

```
timer args
```

- *args* = one or more of *off* or *loop* or *normal* or *full* or *sync* or *nosync* or *timeout* or *every*

off = do not collect or print any timing information
loop = collect only the total time for the simulation loop
normal = collect timer information broken down by sections (default)
full = like normal but also include CPU and thread utilization
sync = explicitly synchronize MPI tasks between sections
nosync = do not synchronize MPI tasks between sections (default)
timeout elapse = set wall time limit to elapse
every Ncheck = perform timeout check every Ncheck steps

1.108.2 Examples

```
timer full sync
timer timeout 2:00:00 every 100
timer loop
```

1.108.3 Description

Select the level of detail at which LAMMPS performs its CPU timings. Multiple keywords can be specified with the *timer* command. For keywords that are mutually exclusive, the last one specified takes precedence.

During a simulation run LAMMPS collects information about how much time is spent in different sections of the code and thus can provide information for determining performance and load imbalance problems. This can be done at different levels of detail and accuracy. For more information about the timing output, see the *Run output* doc page.

The *off* setting will turn all time measurements off. The *loop* setting will only measure the total time for a run and not collect any detailed per section information. With the *normal* setting, timing information for portions of the timestep (pairwise calculations, neighbor list construction, output, etc) are collected as well as information about load imbalances for those sections across processors. The *full* setting adds information about CPU utilization and thread utilization, when multi-threading is enabled.

With the *sync* setting, all MPI tasks are synchronized at each timer call which measures load imbalance for each section more accurately, though it can also slow down the simulation by prohibiting overlapping independent computations on different MPI ranks Using the *nosync* setting (which is the default) turns this synchronization off.

With the *timeout* keyword a wall time limit can be imposed, that affects the *run* and *minimize* commands. This can be convenient when calculations have to comply with execution time limits, e.g. when running under a batch system when you want to maximize the utilization of the batch time slot, especially for runs where the time per timestep varies much and thus it becomes difficult to predict how many steps a simulation can perform for a given wall time limit. This also applies for difficult to converge minimizations. The timeout *elapse* value should be somewhat smaller than the maximum wall time requested from the batch system, as there is usually some overhead to launch jobs, and it is advisable to write out a restart after terminating a run due to a timeout.

The timeout timer starts when the command is issued. When the time limit is reached, the run or energy minimization will exit on the next step or iteration that is a multiple of the *Ncheck* value which can be set with the *every* keyword. Default is checking every 10 steps. After the timer timeout has expired all subsequent run or minimize commands in the

input script will be skipped. The remaining time or timer status can be accessed with the *thermo* variable *timeremain*, which will be zero, if the timeout is inactive (default setting), it will be negative, if the timeout time is expired and positive if there is time remaining and in this case the value of the variable are the number of seconds remaining.

When the *timeout* key word is used a second time, the timer is restarted with a new time limit. The timeout *elapse* value can be specified as *off* or *unlimited* to impose a no timeout condition (which is the default). The *elapse* setting can be specified as a single number for seconds, two numbers separated by a colon (MM:SS) for minutes and seconds, or as three numbers separated by colons for hours, minutes, and seconds (H:MM:SS).

The *every* keyword sets how frequently during a run or energy minimization the wall clock will be checked. This check count applies to the outer iterations or time steps during minimizations or *r-RESPA runs*, respectively. Checking for timeout too often, can slow a calculation down. Checking too infrequently can make the timeout measurement less accurate, with the run being stopped later than desired.

Note

Using the *full* and *sync* options provides the most detailed and accurate timing information, but can also have a negative performance impact due to the overhead of the many required system calls. It is thus recommended to use these settings only when testing tests to identify performance bottlenecks. For calculations with few atoms or a very large number of processors, even the *normal* setting can have a measurable negative performance impact. In those cases you can just use the *loop* or *off* setting.

1.108.4 Restrictions

none

1.108.5 Related commands

run post no, *kspace_modify fftbench*

1.108.6 Default

```
timer normal nosync
timer timeout off
timer every 10
```

1.109 timestep command

1.109.1 Syntax

```
timestep dt
```

- dt = timestep size (time units)

1.109.2 Examples

```
timestep 2.0
timestep 0.003
```

1.109.3 Description

Set the timestep size for subsequent molecular dynamics simulations. See the *units* command for the time units associated with each choice of units that LAMMPS supports.

The default value for the timestep size also depends on the choice of units for the simulation; see the default values below.

When the *run style* is *respa*, dt is the timestep for the outer loop (largest) timestep.

1.109.4 Restrictions

none

1.109.5 Related commands

fix dt/reset, *run*, *run_style* respa, *units*

1.109.6 Default

choice of *units*	time units	default timestep size
lj	τ	0.005 τ
real	fs	1.0 fs
metal	ps	0.001 ps
si	s	1.0e-8 s (10 ns)
cgs	s	1.0e-8 s (10 ns)
electron	fs	0.001 fs
micro	μs	2.0 μs
nano	ns	0.00045 ns

1.110 uncompute command

1.110.1 Syntax

```
uncompute compute-ID
```

- compute-ID = ID of a previously defined compute

1.110.2 Examples

```
uncompute 2
uncompute lower-boundary
```

1.110.3 Description

Delete a compute that was previously defined with a *compute* command. This also wipes out any additional changes made to the compute via the *compute_modify* command.

1.110.4 Restrictions

none

1.110.5 Related commands

compute

1.110.6 Default

none

1.111 undump command

1.111.1 Syntax

```
undump dump-ID
```

- dump-ID = ID of previously defined dump

1.111.2 Examples

```
undump mine
undump 2
```

1.111.3 Description

Turn off a previously defined dump so that it is no longer active. This closes the file associated with the dump.

1.111.4 Restrictions

none

1.111.5 Related commands

dump

1.111.6 Default

none

1.112 unfix command

1.112.1 Syntax

```
unfix fix-ID
```

- fix-ID = ID of a previously defined fix

1.112.2 Examples

```
unfix 2
unfix lower-boundary
```

1.112.3 Description

Delete a fix that was previously defined with a *fix* command. This also wipes out any additional changes made to the fix via the *fix_modify* command.

1.112.4 Restrictions

none

1.112.5 Related commands

fix

1.112.6 Default

none

1.113 units command

1.113.1 Syntax

```
units style
```

- style = *lj* or *real* or *metal* or *si* or *cgs* or *electron* or *micro* or *nano*

1.113.2 Examples

```
units metal
units lj
```

1.113.3 Description

This command sets the style of units used for a simulation. It determines the units of all quantities specified in the input script and data file, as well as quantities output to the screen, log file, and dump files. Typically, this command is used at the very beginning of an input script.

For all units except *lj*, LAMMPS uses physical constants from www.physics.nist.gov. For the definition of kcal in real units, LAMMPS uses the thermochemical calorie = 4.184 J.

The choice you make for units simply sets some internal conversion factors within LAMMPS. This means that any simulation you perform for one choice of units can be duplicated with any other unit setting LAMMPS supports. In this context "duplicate" means the particles will have identical trajectories and all output generated by the simulation will be identical. This will be the case for some number of timesteps until round-off effects accumulate, since the conversion factors for two different unit systems are not identical to infinite precision.

To perform the same simulation in a different set of units you must change all the unit-based input parameters in your input script and other input files (data file, potential files, etc) correctly to the new units. And you must correctly convert all output from the new units to the old units when comparing to the original results. That is often not simple to do.

Potential or table files may have a UNITS: tag included in the first line indicating the unit style those files were created for. If the tag exists, its value will be compared to the chosen unit style and LAMMPS will stop with an error message if there is a mismatch. In some select cases and for specific combinations of unit styles, LAMMPS is capable of automatically converting potential parameters from a file. In those cases, a warning message signaling that an automatic conversion has happened is printed to the screen.

For style *lj*, all quantities are unitless. Without loss of generality, LAMMPS sets the fundamental quantities mass, σ, ϵ, and the Boltzmann constant $k_B = 1$. The masses, distances, energies you specify are multiples of these fundamental values. The formulas relating the reduced or unitless quantity (with an asterisk) to the same quantity with units is also given. Thus you can use the mass, σ, and ϵ values for a specific material and convert the results from a unitless LJ simulation into physical quantities. Please note that using these three properties as base, your unit of time has to conform to the relation $\epsilon = \frac{m\sigma^2}{\tau^2}$ since energy is a derived unit (in SI units you equivalently have the relation $1\,\mathrm{J} = 1\frac{\mathrm{kg}\cdot\mathrm{m}^2}{\mathrm{s}^2}$).

- mass = mass or m, where $M^* = \frac{M}{m}$

- distance = σ, where $x^* = \frac{x}{\sigma}$
- time = τ, where $\tau^* = \tau\sqrt{\frac{\varepsilon}{m\sigma^2}}$
- energy = ε, where $E^* = \frac{E}{\varepsilon}$
- velocity = $\frac{\sigma}{\tau}$, where $v^* = v\frac{\tau}{\sigma}$
- force = $\frac{\varepsilon}{\sigma}$, where $f^* = f\frac{\sigma}{\varepsilon}$
- torque = ε, where $t^* = \frac{t}{\varepsilon}$
- temperature = reduced LJ temperature, where $T^* = \frac{Tk_B}{\varepsilon}$
- pressure = reduced LJ pressure, where $p^* = p\frac{\sigma^3}{\varepsilon}$
- dynamic viscosity = reduced LJ viscosity, where $\eta^* = \eta\frac{\sigma^3}{\varepsilon\tau}$
- charge = reduced LJ charge, where $q^* = q\frac{1}{\sqrt{4\pi\varepsilon_0\sigma\varepsilon}}$
- dipole = reduced LJ dipole, moment where $\mu^* = \mu\frac{1}{\sqrt{4\pi\varepsilon_0\sigma^3\varepsilon}}$
- electric field = force/charge, where $E^* = E\frac{\sqrt{4\pi\varepsilon_0\sigma\varepsilon}\sigma}{\varepsilon}$
- density = mass/volume, where $\rho^* = \rho\frac{\sigma^{dim}}{m}$

Note that for LJ units, the default mode of thermodynamic output via the *thermo_style* command is to normalize all extensive quantities by the number of atoms. E.g. potential energy is extensive because it is summed over atoms, so it is output as energy/atom. Temperature is intensive since it is already normalized by the number of atoms, so it is output as-is. This behavior can be changed via the *thermo_modify norm* command.

For style *real*, these are the units:

- mass = grams/mole
- distance = Angstroms
- time = femtoseconds
- energy = kcal/mol
- velocity = Angstroms/femtosecond
- force = (kcal/mol)/Angstrom
- torque = kcal/mol
- temperature = Kelvin
- pressure = atmospheres
- dynamic viscosity = Poise
- charge = multiple of electron charge (1.0 is a proton)
- dipole = charge*Angstroms
- electric field = volts/Angstrom
- density = g/cm^dim

For style *metal*, these are the units:

- mass = grams/mole
- distance = Angstroms

- time = picoseconds
- energy = eV
- velocity = Angstroms/picosecond
- force = eV/Angstrom
- torque = eV
- temperature = Kelvin
- pressure = bars
- dynamic viscosity = Poise
- charge = multiple of electron charge (1.0 is a proton)
- dipole = charge*Angstroms
- electric field = volts/Angstrom
- density = gram/cm^dim

For style *si*, these are the units:

- mass = kilograms
- distance = meters
- time = seconds
- energy = Joules
- velocity = meters/second
- force = Newtons
- torque = Newton-meters
- temperature = Kelvin
- pressure = Pascals
- dynamic viscosity = Pascal*second
- charge = Coulombs (1.6021765e-19 is a proton)
- dipole = Coulombs*meters
- electric field = volts/meter
- density = kilograms/meter^dim

For style *cgs*, these are the units:

- mass = grams
- distance = centimeters
- time = seconds
- energy = ergs
- velocity = centimeters/second
- force = dynes
- torque = dyne-centimeters
- temperature = Kelvin

- pressure = dyne/cm^2 or barye = 1.0e-6 bars
- dynamic viscosity = Poise
- charge = statcoulombs or esu (4.8032044e-10 is a proton)
- dipole = statcoul-cm = 10^18 debye
- electric field = statvolt/cm or dyne/esu
- density = grams/cm^dim

For style *electron*, these are the units:

- mass = atomic mass units
- distance = Bohr
- time = femtoseconds
- energy = Hartrees
- velocity = Bohr/atomic time units [1.03275e-15 seconds]
- force = Hartrees/Bohr
- temperature = Kelvin
- pressure = Pascals
- charge = multiple of electron charge (1.0 is a proton)
- dipole moment = Debye
- electric field = volts/cm

For style *micro*, these are the units:

- mass = picograms
- distance = micrometers
- time = microseconds
- energy = picogram-micrometer^2/microsecond^2
- velocity = micrometers/microsecond
- force = picogram-micrometer/microsecond^2
- torque = picogram-micrometer^2/microsecond^2
- temperature = Kelvin
- pressure = picogram/(micrometer-microsecond^2)
- dynamic viscosity = picogram/(micrometer-microsecond)
- charge = picocoulombs (1.6021765e-7 is a proton)
- dipole = picocoulomb-micrometer
- electric field = volt/micrometer
- density = picograms/micrometer^dim

For style *nano*, these are the units:

- mass = attograms
- distance = nanometers

- time = nanoseconds
- energy = attogram-nanometer^2/nanosecond^2
- velocity = nanometers/nanosecond
- force = attogram-nanometer/nanosecond^2
- torque = attogram-nanometer^2/nanosecond^2
- temperature = Kelvin
- pressure = attogram/(nanometer-nanosecond^2)
- dynamic viscosity = attogram/(nanometer-nanosecond)
- charge = multiple of electron charge (1.0 is a proton)
- dipole = charge-nanometer
- electric field = volt/nanometer
- density = attograms/nanometer^dim

The units command also sets the timestep size and neighbor skin distance to default values for each style:

- For style *lj* these are dt = 0.005 τ and skin = 0.3 σ.
- For style *real* these are dt = 1.0 femtoseconds and skin = 2.0 Angstroms.
- For style *metal* these are dt = 0.001 picoseconds and skin = 2.0 Angstroms.
- For style *si* these are dt = 1.0e-8 seconds and skin = 0.001 meters.
- For style *cgs* these are dt = 1.0e-8 seconds and skin = 0.1 centimeters.
- For style *electron* these are dt = 0.001 femtoseconds and skin = 2.0 Bohr.
- For style *micro* these are dt = 2.0 microseconds and skin = 0.1 micrometers.
- For style *nano* these are dt = 0.00045 nanoseconds and skin = 0.1 nanometers.

1.113.4 Restrictions

This command cannot be used after the simulation box is defined by a *read_data* or *create_box* command.

1.113.5 Related commands

none

1.113.6 Default

```
units lj
```

1.114 variable command

1.114.1 Syntax

```
variable name style args ...
```

- name = name of variable to define
- style = *delete* or *atomfile* or *file* or *format* or *getenv* or *index* or *internal* or *loop* or *python* or *string* or *timer* or *uloop* or *universe* or *world* or *equal* or *vector* or *atom*

```
delete = no args
atomfile arg = filename
file arg = filename
format args = vname fstr
  vname = name of equal-style variable to evaluate
  fstr = C-style format string
getenv arg = one string
index args = one or more strings
internal arg = numeric value
loop args = N
  N = integer size of loop, loop from 1 to N inclusive
loop args = N pad
  N = integer size of loop, loop from 1 to N inclusive
  pad = all values will be same length, e.g. 001, 002, ..., 100
loop args = N1 N2
  N1,N2 = loop from N1 to N2 inclusive
loop args = N1 N2 pad
  N1,N2 = loop from N1 to N2 inclusive
  pad = all values will be same length, e.g. 050, 051, ..., 100
python arg = function
string arg = one string
timer arg = no arguments
uloop args = N
  N = integer size of loop
uloop args = N pad
  N = integer size of loop
  pad = all values will be same length, e.g. 001, 002, ..., 100
universe args = one or more strings
world args = one string for each partition of processors

equal or vector or atom args = one formula containing numbers, thermo keywords, math operations,
 ↪ built-in functions, atom values and vectors, compute/fix/variable references
  numbers = 0.0, 100, -5.4, 2.8e-4, etc
  constants = PI, version, on, off, true, false, yes, no
  thermo keywords = vol, ke, press, etc from thermo_style
  math operators = (), -x, x+y, x-y, x*y, x/y, x^y, x%y,
                   x == y, x != y, x < y, x <= y, x > y, x >= y, x && y, x || y, x |^ y, !x
  math functions = sqrt(x), exp(x), ln(x), log(x), abs(x),
                   sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), atan2(y,x),
                   random(x,y,z), normal(x,y,z), ceil(x), floor(x), round(x), ternary(x,y,z),
                   ramp(x,y), stagger(x,y), logfreq(x,y,z), logfreq2(x,y,z),
                   logfreq3(x,y,z), stride(x,y,z), stride2(x,y,z,a,b,c),
                   vdisplace(x,y), swiggle(x,y,z), cwiggle(x,y,z)
```

```
group functions = count(group), mass(group), charge(group),
                  xcm(group,dim), vcm(group,dim), fcm(group,dim),
                  bound(group,dir), gyration(group), ke(group),
                  angmom(group,dim), torque(group,dim),
                  inertia(group,dimdim), omega(group,dim)
region functions = count(group,region), mass(group,region), charge(group,region),
                  xcm(group,dim,region), vcm(group,dim,region), fcm(group,dim,region),
                  bound(group,dir,region), gyration(group,region), ke(group,reigon),
                  angmom(group,dim,region), torque(group,dim,region),
                  inertia(group,dimdim,region), omega(group,dim,region)
special functions = sum(x), min(x), max(x), ave(x), trap(x), slope(x), sort(x), rsort(x),
↪gmask(x), rmask(x), grmask(x,y), next(x), is_file(name), is_os(name), extract_setting(name),
↪ label2type(kind,label), is_typelabel(kind,label), is_timeout()
feature functions = is_available(category,feature), is_active(category,feature), is_defined(category,
↪id)
atom value = id[i], mass[i], type[i], mol[i], x[i], y[i], z[i], vx[i], vy[i], vz[i], fx[i], fy[i], fz[i], q[i]
atom vector = id, mass, type, mol, radius, q, x, y, z, vx, vy, vz, fx, fy, fz
custom atom property = i_name, d_name, i_name[i], d_name[i], i2_name[i], d2_name[i], i2_
↪name[i][j], d_name[i][j]
compute references = c_ID, c_ID[i], c_ID[i][j], C_ID, C_ID[i]
fix references = f_ID, f_ID[i], f_ID[i][j], F_ID, F_ID[i]
variable references = v_name, v_name[i]
vector initialization = [1,3,7,10] (for vector variables only)
```

1.114.2 Examples

```
variable x index run1 run2 run3 run4 run5 run6 run7 run8
variable LoopVar loop $n
variable beta equal temp/3.0
variable b1 equal x[234]+0.5*vol
variable b1 equal "x[234] + 0.5*vol"
variable b equal xcm(mol1,x)/2.0
variable b equal c_myTemp
variable b atom x*y/vol
variable foo string myfile
variable foo internal 3.5
variable myPy python increase
variable f file values.txt
variable temp world 300.0 310.0 320.0 ${Tfinal}
variable x universe 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
variable x uloop 15 pad
variable str format x %.6g
variable myvec vector [1,3,7,10]
variable x delete
```

```
variable start timer
other commands
variable stop timer
print "Elapsed time: $(v_stop-v_start:%.6f)"
```

1.114.3 Description

This command assigns one or more strings to a variable name for evaluation later in the input script or during a simulation.

Variables can thus be useful in several contexts. A variable can be defined and then referenced elsewhere in an input script to become part of a new input command. For variable styles that store multiple strings, the *next* command can be used to increment which string is assigned to the variable. Variables of style *equal* store a formula which when evaluated produces a single numeric value which can be output either directly (see the *print*, *fix print*, and *run every* commands) or as part of thermodynamic output (see the *thermo_style* command), or used as input to an averaging fix (see the *fix ave/time* command). Variables of style *vector* store a formula which produces a vector of such values which can be used as input to various averaging fixes, or elements of which can be part of thermodynamic output. Variables of style *atom* store a formula which when evaluated produces one numeric value per atom which can be output to a dump file (see the *dump custom* command) or used as input to an averaging fix (see the *fix ave/chunk* and *fix ave/atom* commands). Variables of style *atomfile* can be used anywhere in an input script that atom-style variables are used; they get their per-atom values from a file rather than from a formula. Variables of style *python* can be hooked to Python functions using code you provide, so that the variable gets its value from the evaluation of the Python code. Variables of style *internal* are used by a few commands which set their value directly.

Note

As discussed on the *Commands parse* doc page, an input script can use "immediate" variables, specified as $(formula) with parenthesis, where the numeric formula has the same syntax as equal-style variables described on this page. This is a convenient way to evaluate a formula immediately without using the variable command to define a named variable and then evaluate that variable. The formula can include a trailing colon and format string which determines the precision with which the numeric value is generated. This is also explained on the *Commands parse* doc page.

In the discussion that follows, the "name" of the variable is the arbitrary string that is the first argument in the variable command. This name can only contain alphanumeric characters and underscores. The "string" is one or more of the subsequent arguments. The "string" can be simple text as in the first example above, it can contain other variables as in the second example, or it can be a formula as in the third example. The "value" is the numeric quantity resulting from evaluation of the string. Note that the same string can generate different values when it is evaluated at different times during a simulation.

Note

When an input script line is encountered that defines a variable of style *equal* or *vector* or *atom* or *python* that contains a formula or Python code, the formula is NOT immediately evaluated. It will be evaluated every time when the variable is **used** instead. If you simply want to evaluate a formula in place you can use as so-called. See the section below about "Immediate Evaluation of Variables" for more details on the topic. This is also true of a *format* style variable since it evaluates another variable when it is invoked.

Variables of style *equal* and *vector* and *atom* can be used as inputs to various other commands which evaluate their formulas as needed, e.g. at different timesteps during a *run*. In this context, variables of style *timer* or *internal* or *python* can be used in place of an equal-style variable, with the following two caveats.

First, internal-style variables can be used except by commands that set the value stored by the internal variable. When the LAMMPS command evaluates the internal-style variable, it will use the value set (internally) by another command. Second, python-style variables can be used so long as the associated Python function, as defined by the *python* command, returns a numeric value. When the LAMMPS command evaluates the python-style variable, the Python function will be executed.

Note

When a variable command is encountered in the input script and the variable name has already been specified, the command is ignored. This means variables can NOT be re-defined in an input script (with two exceptions, read further). This is to allow an input script to be processed multiple times without resetting the variables; see the *jump* or *include* commands. It also means that using the *command-line switch* -var will override a corresponding index variable setting in the input script.

There are two exceptions to this rule. First, variables of style *string*, *getenv*, *internal*, *equal*, *vector*, *atom*, and *python* ARE redefined each time the command is encountered. This allows these style of variables to be redefined multiple times in an input script. In a loop, this means the formula associated with an *equal* or *atom* style variable can change if it contains a substitution for another variable, e.g. $x or v_x.

Second, as described below, if a variable is iterated on to the end of its list of strings via the *next* command, it is removed from the list of active variables, and is thus available to be re-defined in a subsequent variable command. The *delete* style does the same thing.

Variables are **not** deleted by the *clear* command with the exception of atomfile-style variables.

The *Commands parse* page explains how occurrences of a variable name in an input script line are replaced by the variable's string. The variable name can be referenced as $x if the name "x" is a single character, or as ${LoopVar} if the name "LoopVar" is one or more characters.

As described below, for variable styles *index*, *loop*, *file*, *universe*, and *uloop*, which string is assigned to a variable can be incremented via the *next* command. When there are no more strings to assign, the variable is exhausted and a flag is set that causes the next *jump* command encountered in the input script to be skipped. This enables the construction of simple loops in the input script that are iterated over and then exited from.

As explained above, an exhausted variable can be re-used in an input script. The *delete* style also removes the variable, the same as if it were exhausted, allowing it to be redefined later in the input script or when the input script is looped over. This can be useful when breaking out of a loop via the *if* and *jump* commands before the variable would become exhausted. For example,

```
label     loop
variable   a loop 5
print      "A = $a"
if        "$a > 2" then "jump in.script break"
next       a
jump       in.script loop
label     break
variable   a delete
```

The next sections describe in how all the various variable styles are defined and what they store. The styles are listed alphabetically, except for the *equal* and *vector* and *atom* styles, which are explained together after all the others.

Many of the styles store one or more strings. Note that a single string can contain spaces (multiple words), if it is enclosed in quotes in the variable command. When the variable is substituted for in another input script command, its returned string will then be interpreted as multiple arguments in the expanded command.

For the *atomfile* style, a filename is provided which contains one or more sets of values, to assign on a per-atom basis to the variable. The format of the file is described below.

When an atomfile-style variable is defined, the file is opened and the first set of per-atom values are read and stored with the variable. This means the variable can then be evaluated as many times as desired and will return those values. There are two ways to cause the next set of per-atom values from the file to be read: use the *next* command or the next() function in an atom-style variable, as discussed below. Unlike most variable styles, which remain defined, atomfile-style variables are **deleted** during a *clear* command.

The rules for formatting the file are as follows. Each time a set of per-atom values is read, a non-blank line is searched for in the file. The file is read line by line but only up to 254 characters are used. The rest are ignored. A comment character "#" can be used anywhere on a line and all text following and the "#" character are ignored; text starting with the comment character is stripped. Blank lines are skipped. The first non-blank line is expected to contain a single integer number as the count N of per-atom lines to follow. N can be the total number of atoms in the system or less, indicating that data for a subset is read. The next N lines must consist of two numbers, the atom-ID of the atom for which a value is set followed by a floating point number with the value. The atom-IDs may be listed in any order.

Note

Every time a set of per-atom lines is read, the value of the atomfile variable for **all** atoms is first initialized to 0.0. Thus values for atoms whose ID do not appear in the set in the file will remain at 0.0.

Below is a small example for the atomfile variable file format:

```
# first set
4
# atom-ID value
3 1
4 -4
1 0.5
2 -0.5

# second set
2

2  1.0
4 -1.0
```

For the *file* style, a filename is provided which contains a list of strings to assign to the variable, one per line. The strings can be numeric values if desired. See the discussion of the next() function below for equal-style variables, which will convert the string of a file-style variable into a numeric value in a formula.

When a file-style variable is defined, the file is opened and the string on the first line is read and stored with the variable. This means the variable can then be evaluated as many times as desired and will return that string. There are two ways to cause the next string from the file to be read: use the *next* command or the next() function in an equal- or atom-style variable, as discussed below.

The rules for formatting the file are as follows. A comment character "#" can be used anywhere on a line; text starting with the comment character is stripped. Blank lines are skipped. The first "word" of a non-blank line, delimited by white-space, is the "string" assigned to the variable.

For the *format* style, an equal-style or compatible variable is specified along with a C-style format string, e.g. "%f" or "%.10g", which must be appropriate for formatting a double-precision floating-point value and may not have extra characters. The default format is "%.15g". This variable style allows an equal-style variable to be formatted precisely when it is evaluated.

Note that if you simply wish to print a variable value with desired precision to the screen or logfile via the *print* or *fix print* commands, you can also do this by specifying an "immediate" variable with a trailing colon and format string, as part of the string argument of those commands. This is explained on the *Commands parse* doc page.

For the *getenv* style, a single string is assigned to the variable which should be the name of an environment variable. When the variable is evaluated, it returns the value of the environment variable, or an empty string if it not defined. This style of variable can be used to adapt the behavior of LAMMPS input scripts via environment variable settings, or to retrieve information that has been previously stored with the *shell putenv* command. Note that because environment variable settings are stored by the operating systems, they persist even if the corresponding *getenv* style variable is deleted, and also are set for sub-shells executed by the *shell* command.

For the *index* style, one or more strings are specified. Initially, the first string is assigned to the variable. Each time a *next* command is used with the variable name, the next string is assigned. All processors assign the same string to the variable.

Index-style variables with a single string value can also be set by using the *command-line switch -var*.

For the *internal* style a numeric value is provided. This value will be assigned to the variable until a LAMMPS command sets it to a new value. There are currently only two LAMMPS commands that require *internal* variables as inputs, because they reset them: *create_atoms* and *fix controller*. As mentioned above, an internal-style variable can be used in place of an equal-style variable anywhere else in an input script, e.g. as an argument to another command that allows for equal-style variables.

The *loop* style is identical to the *index* style except that the strings are the integers from 1 to N inclusive, if only one argument N is specified. This allows generation of a long list of runs (e.g. 1000) without having to list N strings in the input script. Initially, the string "1" is assigned to the variable. Each time a *next* command is used with the variable name, the next string ("2", "3", etc) is assigned. All processors assign the same string to the variable. The *loop* style can also be specified with two arguments N1 and N2. In this case the loop runs from N1 to N2 inclusive, and the string N1 is initially assigned to the variable. N1 <= N2 and N2 >= 0 is required.

For the *python* style a Python function name is provided. This needs to match a function name specified in a *python* command which returns a value to this variable as defined by its *return* keyword. For example these two commands would be self-consistent:

```
variable foo python myMultiply
python myMultiply return v_foo format f file funcs.py
```

The two commands can appear in either order so long as both are specified before the Python function is invoked for the first time.

Each time the variable is evaluated, the associated Python function is invoked, and the value it returns is also returned by the variable. Since the Python function can use other LAMMPS variables as input, or query interal LAMMPS quantities to perform its computation, this means the variable can return a different value each time it is evaluated.

The type of value stored in the variable is determined by the *format* keyword of the *python* command. It can be an integer (i), floating point (f), or string (s) value. As mentioned above, if it is a numeric value (integer or floating point), then the python-style variable can be used in place of an equal-style variable anywhere in an input script, e.g. as an argument to another command that allows for equal-style variables.

For the *string* style, a single string is assigned to the variable. Two differences between this style and using the *index* style exist: a variable with *string* style can be redefined, e.g. by another command later in the input script, or if the script is read again in a loop. The other difference is that *string* performs variable substitution even if the string parameter is quoted.

The *uloop* style is identical to the *universe* style except that the strings are the integers from 1 to N. This allows generation of long list of runs (e.g. 1000) without having to list N strings in the input script.

For the *universe* style, one or more strings are specified. There must be at least as many strings as there are processor partitions or "worlds". LAMMPS can be run with multiple partitions via the *-partition command-line switch*. This variable command initially assigns one string to each world. When a *next* command is encountered using this variable, the first processor partition to encounter it, is assigned the next available string. This continues until all the variable strings are consumed. Thus, this command can be used to run 50 simulations on 8 processor partitions. The simulations will be run one after the other on whatever partition becomes available, until they are all finished. Universe-style variables are incremented using the files "tmp.lammps.variable" and "tmp.lammps.variable.lock" which you will see in your directory during such a LAMMPS run.

For the *world* style, one or more strings are specified. There must be one string for each processor partition or "world". LAMMPS can be run with multiple partitions via the *-partition command-line switch*. This variable command assigns one string to each world. All processors in the world are assigned the same string. The next command cannot be used with equal-style variables, since there is only one value per world. This style of variable is useful when you wish to run different simulations on different partitions, or when performing a parallel tempering simulation (see the *temper* command), to assign different temperatures to different partitions.

For the *equal* and *vector* and *atom* styles, a single string is specified which represents a formula that will be evaluated afresh each time the variable is used. If you want spaces in the string, enclose it in double quotes so the parser will treat it as a single argument. For *equal*-style variables the formula computes a scalar quantity, which becomes the value of the variable whenever it is evaluated. For *vector*-style variables the formula must compute a vector of quantities, which becomes the value of the variable whenever it is evaluated. The calculated vector can be of length one, but it cannot be a simple scalar value like that produced by an equal-style compute. I.e. the formula for a vector-style variable must have at least one quantity in it that refers to a global vector produced by a compute, fix, or other vector-style variable. For *atom*-style variables the formula computes one quantity for each atom whenever it is evaluated.

Note that *equal*, *vector*, and *atom* variables can produce different values at different stages of the input script or at different times during a run. For example, if an *equal* variable is used in a *fix print* command, different values could be printed each timestep it was invoked. If you want a variable to be evaluated immediately, so that the result is stored by the variable instead of the string, see the section below on "Immediate Evaluation of Variables".

The next command cannot be used with *equal* or *vector* or *atom* style variables, since there is only one string.

The formula for an *equal*, *vector*, or *atom* variable can contain a variety of quantities. The syntax for each kind of quantity is simple, but multiple quantities can be nested and combined in various ways to build up formulas of arbitrary complexity. For example, this is a valid (though strange) variable formula:

```
variable x equal "pe + c_MyTemp / vol^(1/3)"
```

Specifically, a formula can contain numbers, constants, thermo keywords, math operators, math functions, group functions, region functions, special functions, feature functions, atom values, atom vectors, custom atom properties, compute references, fix references, and references to other variables.

Number	0.2, 100, 1.0e20, -15.4, etc
Constant	PI, version, on, off, true, false, yes, no
Thermo keywords	vol, pe, ebond, etc
Math operators	(), -x, x+y, x-y, x*y, x/y, x^y, x%y, x == y, x != y, x < y, x <= y, x > y, x >= y, x && y, x || y, x |^ y, !x
Math functions	sqrt(x), exp(x), ln(x), log(x), abs(x), sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), atan2(y,x), random(x,y,z), normal(x,y,z), ceil(x), floor(x), round(x), ternary(x,y,z), ramp(x,y), stagger(x,y), logfreq(x,y,z), logfreq2(x,y,z), logfreq3(x,y,z), stride(x,y,z), stride2(x,y,z,a,b,c), vdisplace(x,y), swiggle(x,y,z), cwiggle(x,y,z)
Group functions	count(ID), mass(ID), charge(ID), xcm(ID,dim), vcm(ID,dim), fcm(ID,dim), bound(ID,dir), gyration(ID), ke(ID), angmom(ID,dim), torque(ID,dim), inertia(ID,dimdim), omega(ID,dim)
Region functions	count(ID,IDR), mass(ID,IDR), charge(ID,IDR), xcm(ID,dim,IDR), vcm(ID,dim,IDR), fcm(ID,dim,IDR), bound(ID,dir,IDR), gyration(ID,IDR), ke(ID,IDR), angmom(ID,dim,IDR), torque(ID,dim,IDR), inertia(ID,dimdim,IDR), omega(ID,dim,IDR)
Special functions	sum(x), min(x), max(x), ave(x), trap(x), slope(x), sort(x), rsort(x), gmask(x), rmask(x), grmask(x,y), next(x), is_file(name), is_os(name), extract_setting(name), label2type(kind,label), is_typelabel(kind,label), is_timeout()
Feature functions	is_available(category,feature), is_active(category,feature), is_defined(category,id)
Atom values	id[i], mass[i], type[i], mol[i], x[i], y[i], z[i], vx[i], vy[i], vz[i], fx[i], fy[i], fz[i], q[i]
Atom vectors	id, mass, type, mol, x, y, z, vx, vy, vz, fx, fy, fz, q
Custom atom properties	i_name, d_name, i_name[i], d_name[i], i2_name[i], d2_name[i], i2_name[i][j], d_name[i][j]
Compute references	c_ID, c_ID[i], c_ID[i][j], C_ID, C_ID[i]
Fix references	f_ID, f_ID[i], f_ID[i][j], F_ID, F_ID[i]
Other variables	v_name, v_name[i]

Most of the formula elements produce a scalar value. Some produce a global or per-atom vector of values. Global vectors can be produced by computes or fixes or by other vector-style variables. Per-atom vectors are produced by atom vectors, computes or fixes which output a per-atom vector or array, and variables that are atom-style variables. Math functions that operate on scalar values produce a scalar value; math function that operate on global or per-atom vectors do so element-by-element and produce a global or per-atom vector.

A formula for equal-style variables cannot use any formula element that produces a global or per-atom vector. A formula for a vector-style variable can use formula elements that produce either a scalar value or a global vector value, but cannot use a formula element that produces a per-atom vector. A formula for an atom-style variable can use formula

elements that produce either a scalar value or a per-atom vector, but not one that produces a global vector.

Atom-style variables are evaluated by other commands that define a *group* on which they operate, e.g. a *dump* or *compute* or *fix* command. When they invoke the atom-style variable, only atoms in the group are included in the formula evaluation. The variable evaluates to 0.0 for atoms not in the group.

Numbers, constants, and thermo keywords

Numbers can contain digits, scientific notation (3.0e20,3.0e-20,3.0E20,3.0E-20), and leading minus signs.

Constants are set at compile time and cannot be changed. *PI* will return the number 3.14159265358979323846; *on*, *true* or *yes* will return 1.0; *off*, *false* or *no* will return 0.0; *version* will return a numeric version code of the current LAMMPS version (e.g. version 2 Sep 2015 will return the number 20150902). The corresponding value for newer versions of LAMMPS will be larger, for older versions of LAMMPS will be smaller. This can be used to have input scripts adapt automatically to LAMMPS versions, when non-backwards compatible syntax changes are introduced. Here is an illustrative example (which will not work, since the *version* has been introduced more recently):

```
if $(version<20140513) then "communicate vel yes" else "comm_modify vel yes"
```

The thermo keywords allowed in a formula are those defined by the *thermo_style custom* command. Thermo keywords that require a *compute* to calculate their values such as "temp" or "press", use computes stored and invoked by the *thermo_style* command. This means that you can only use those keywords in a variable if the style you are using with the thermo_style command (and the thermo keywords associated with that style) also define and use the needed compute. Note that some thermo keywords use a compute indirectly to calculate their value (e.g. the enthalpy keyword uses temp, pe, and pressure). If a variable is evaluated directly in an input script (not during a run), then the values accessed by the thermo keyword must be current. See the discussion below about "Variable Accuracy".

Math Operators

Math operators are written in the usual way, where the "x" and "y" in the examples can themselves be arbitrarily complex formulas, as in the examples above. In this syntax, "x" and "y" can be scalar values or per-atom vectors. For example, "ke/natoms" is the division of two scalars, where "vy+vz" is the element-by-element sum of two per-atom vectors of y and z velocities.

Operators are evaluated left to right and have the usual C-style precedence: unary minus and unary logical NOT operator "!" have the highest precedence, exponentiation "^" is next; multiplication and division and the modulo operator "%" are next; addition and subtraction are next; the 4 relational operators "<", "<=", ">", and ">=" are next; the two remaining relational operators "==" and "!=" are next; then the logical AND operator "&&"; and finally the logical OR operator "||" and logical XOR (exclusive or) operator "|^" have the lowest precedence. Parenthesis can be used to group one or more portions of a formula and/or enforce a different order of evaluation than what would occur with the default precedence.

> **Note**
>
> Because a unary minus is higher precedence than exponentiation, the formula "-2^2" will evaluate to 4, not -4. This convention is compatible with some programming languages, but not others. As mentioned, this behavior can be easily overridden with parenthesis; the formula "-(2^2)" will evaluate to -4.

The 6 relational operators return either a 1.0 or 0.0 depending on whether the relationship between x and y is TRUE or FALSE. For example the expression x<10.0 in an atom-style variable formula will return 1.0 for all atoms whose

x-coordinate is less than 10.0, and 0.0 for the others. The logical AND operator will return 1.0 if both its arguments are non-zero, else it returns 0.0. The logical OR operator will return 1.0 if either of its arguments is non-zero, else it returns 0.0. The logical XOR operator will return 1.0 if one of its arguments is zero and the other non-zero, else it returns 0.0. The logical NOT operator returns 1.0 if its argument is 0.0, else it returns 0.0.

These relational and logical operators can be used as a masking or selection operation in a formula. For example, the number of atoms whose properties satisfy one or more criteria could be calculated by taking the returned per-atom vector of ones and zeroes and passing it to the *compute reduce* command.

Math Functions

Math functions are specified as keywords followed by one or more parenthesized arguments "x", "y", "z", each of which can themselves be arbitrarily complex formulas. In this syntax, the arguments can represent scalar values or global vectors or per-atom vectors. In the latter case, the math operation is performed on each element of the vector. For example, "sqrt(natoms)" is the sqrt() of a scalar, where "sqrt(y*z)" yields a per-atom vector with each element being the sqrt() of the product of one atom's y and z coordinates.

Most of the math functions perform obvious operations. The ln() is the natural log; log() is the base 10 log.

The random(x,y,z) function takes 3 arguments: x = lo, y = hi, and z = seed. It generates a uniform random number between lo and hi. The normal(x,y,z) function also takes 3 arguments: x = mu, y = sigma, and z = seed. It generates a Gaussian variate centered on mu with variance sigma^2. In both cases the seed is used the first time the internal random number generator is invoked, to initialize it. For equal-style and vector-style variables, every processor uses the same seed so that they each generate the same sequence of random numbers. For atom-style variables, a unique seed is created for each processor, based on the specified seed. This effectively generates a different random number for each atom being looped over in the atom-style variable.

Note

Internally, there is just one random number generator for all equal-style and vector-style variables and another one for all atom-style variables. If you define multiple variables (of each style) which use the random() or normal() math functions, then the internal random number generators will only be initialized once, which means only one of the specified seeds will determine the sequence of generated random numbers.

The ceil(), floor(), and round() functions are those in the C math library. Ceil() is the smallest integer not less than its argument. Floor() if the largest integer not greater than its argument. Round() is the nearest integer to its argument.

Added in version 7Feb2024.

The ternary(x,y,z) function is the equivalent of the ternary operator (? and :) in C or C++. It takes 3 arguments. The first argument is a conditional. The result of the function is y if x evaluates to true (non-zero). The result is z if x evaluates to false (zero).

The ramp(x,y) function uses the current timestep to generate a value linearly interpolated between the specified x,y values over the course of a run, according to this formula:

value = x + (y-x) * (timestep-startstep) / (stopstep-startstep)

The run begins on startstep and ends on stopstep. Startstep and stopstep can span multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this. If called in between runs or during a *run 0* command, the ramp(x,y) function will return the value of x.

The stagger(x,y) function uses the current timestep to generate a new timestep. X,y > 0 and x > y are required. The generated timesteps increase in a staggered fashion, as the sequence x,x+y,2x,2x+y,3x,3x+y,etc. For any current timestep,

the next timestep in the sequence is returned. Thus if stagger(1000,100) is used in a variable by the *dump_modify every* command, it will generate the sequence of output timesteps:

```
100,1000,1100,2000,2100,3000,etc
```

The logfreq(x,y,z) function uses the current timestep to generate a new timestep. X,y,z > 0 and y < z are required. The generated timesteps are on a base-z logarithmic scale, starting with x, and the y value is how many of the z-1 possible timesteps within one logarithmic interval are generated. I.e. the timesteps follow the sequence x,2x,3x,...y*x,x*z,2x*z,3x*z,...y*x*z,x*z^2,2x*z^2,etc. For any current timestep, the next timestep in the sequence is returned. Thus if logfreq(100,4,10) is used in a variable by the *dump_modify every* command, it will generate this sequence of output timesteps:

```
100,200,300,400,1000,2000,3000,4000,10000,20000,etc
```

The logfreq2(x,y,z) function is similar to logfreq, except a single logarithmic interval is divided into y equally-spaced timesteps and all of them are output. Y < z is not required. Thus, if logfreq2(100,18,10) is used in a variable by the *dump_modify every* command, then the interval between 100 and 1000 is divided as 900/18 = 50 steps, and it will generate the sequence of output timesteps:

```
100,150,200,...950,1000,1500,2000,...9500,10000,15000,etc
```

The logfreq3(x,y,z) function generates y points between x and z (inclusive), that are separated by a multiplicative ratio: (z/x)^(1/(y-1)). Constraints are: x,z > 0, y > 1, z-x >= y-1. For eg., if logfreq3(10,25,1000) is used in a variable by the *fix print* command, then the interval between 10 and 1000 is divided into 24 parts with a multiplicative separation of ~1.21, and it will generate the following sequence of output timesteps:

```
10, 13, 15, 18, 22, 27, 32,...384, 465, 563, 682, 826, 1000
```

The stride(x,y,z) function uses the current timestep to generate a new timestep. X,y >= 0 and z > 0 and x <= y are required. The generated timesteps increase in increments of z, from x to y, i.e. it generates the sequence x,x+z,x+2z,...,y. If y-x is not a multiple of z, then similar to the way a for loop operates, the last value will be one that does not exceed y. For any current timestep, the next timestep in the sequence is returned. Thus if stride(1000,2000,100) is used in a variable by the *dump_modify every* command, it will generate the sequence of output timesteps:

```
1000,1100,1200, ... ,1900,2000
```

The stride2(x,y,z,a,b,c) function is similar to the stride() function except it generates two sets of strided timesteps, one at a coarser level and one at a finer level. Thus it is useful for debugging, e.g. to produce output every timestep at the point in simulation when a problem occurs. X,y >= 0 and z > 0 and x <= y are required, as are a,b >= 0 and c > 0 and a < b. Also, a >= x and b <= y are required so that the second stride is inside the first. The generated timesteps increase in increments of z, starting at x, until a is reached. At that point the timestep increases in increments of c, from a to b, then after b, increments by z are resumed until y is reached. For any current timestep, the next timestep in the sequence is returned. Thus if stride2(1000,2000,100,1350,1360,1) is used in a variable by the *dump_modify every* command, it will generate the sequence of output timesteps:

```
1000,1100,1200,1300,1350,1351,1352, ... 1359,1360,1400,1500, ... ,2000
```

The vdisplace(x,y) function takes 2 arguments: x = value0 and y = velocity, and uses the elapsed time to change the value by a linear displacement due to the applied velocity over the course of a run, according to this formula:

value = value0 + velocity*(timestep-startstep)*dt

where dt = the timestep size.

The run begins on startstep. Startstep can span multiple runs, using the *start* keyword of the *run* command. See the *run* command for details of how to do this. Note that the *thermo_style* keyword elaplong = timestep-startstep. If used between runs this function will return the value according to the end of the last run or the value of x if used before *any*

runs. This function assumes the length of the time step does not change and thus may not be used in combination with *fix dt/reset*.

The swiggle(x,y,z) and cwiggle(x,y,z) functions each take 3 arguments: x = value0, y = amplitude, z = period. They use the elapsed time to oscillate the value by a sin() or cos() function over the course of a run, according to one of these formulas, where omega = 2 PI / period:

value = value0 + Amplitude * sin(omega*(timestep-startstep)*dt)
value = value0 + Amplitude * (1 - cos(omega*(timestep-startstep)*dt))

where dt = the timestep size.

The run begins on startstep. Startstep can span multiple runs, using the *start* keyword of the *run* command. See the *run* command for details of how to do this. Note that the *thermo_style* keyword elaplong = timestep-startstep. If used between runs these functions will return the value according to the end of the last run or the value of x if used before *any* runs. These functions assume the length of the time step does not change and thus may not be used in combination with *fix dt/reset*.

Group and Region Functions

Group functions are specified as keywords followed by one or two parenthesized arguments. The first argument *ID* is the group-ID. The *dim* argument, if it exists, is *x* or *y* or *z*. The *dir* argument, if it exists, is *xmin*, *xmax*, *ymin*, *ymax*, *zmin*, or *zmax*. The *dimdim* argument, if it exists, is *xx* or *yy* or *zz* or *xy* or *yz* or *xz*.

The group function count() is the number of atoms in the group. The group functions mass() and charge() are the total mass and charge of the group. Xcm() and vcm() return components of the position and velocity of the center of mass of the group. Fcm() returns a component of the total force on the group of atoms. Bound() returns the min/max of a particular coordinate for all atoms in the group. Gyration() computes the radius-of-gyration of the group of atoms. See the *compute gyration* command for a definition of the formula. Angmom() returns components of the angular momentum of the group of atoms around its center of mass. Torque() returns components of the torque on the group of atoms around its center of mass, based on current forces on the atoms. Inertia() returns one of 6 components of the symmetric inertia tensor of the group of atoms around its center of mass, ordered as Ixx,Iyy,Izz,Ixy,Iyz,Ixz. Omega() returns components of the angular velocity of the group of atoms around its center of mass.

Region functions are specified exactly the same way as group functions except they take an extra final argument *IDR* which is the region ID. The function is computed for all atoms that are in both the group and the region. If the group is "all", then the only criteria for atom inclusion is that it be in the region.

Special Functions

Special functions take specific kinds of arguments, meaning their arguments cannot be formulas themselves.

The sum(x), min(x), max(x), ave(x), trap(x), slope(x), sort(x), and rsort(x) functions each take 1 argument which is of the form "c_ID" or "c_ID[N]" or "f_ID" or "f_ID[N]" or "v_name". The first two are computes and the second two are fixes; the ID in the reference should be replaced by the ID of a compute or fix defined elsewhere in the input script. The compute or fix must produce either a global vector or array. If it produces a global vector, then the notation without "[N]" should be used. If it produces a global array, then the notation with "[N]" should be used, where N is an integer, to specify which column of the global array is being referenced. The last form of argument "v_name" is for a vector-style variable where "name" is replaced by the name of the variable.

The sum(x), min(x), max(x), ave(x), trap(x), and slope(x) functions operate on a global vector of inputs and reduce it to a single scalar value. This is analogous to the operation of the *compute reduce* command, which performs similar operations on per-atom and local vectors.

The sort(x) and rsort(x) functions operate on a global vector of inputs and return a global vector of the same length.

The sum() function calculates the sum of all the vector elements. The min() and max() functions find the minimum and maximum element respectively. The ave() function is the same as sum() except that it divides the result by the length of the vector.

The trap() function is the same as sum() except the first and last elements are multiplied by a weighting factor of 1/2 when performing the sum. This effectively implements an integration via the trapezoidal rule on the global vector of data. I.e. consider a set of points, equally spaced by 1 in their x coordinate: (1,V1), (2,V2), ..., (N,VN), where the Vi are the values in the global vector of length N. The integral from 1 to N of these points is trap(). When appropriately normalized by the timestep size, this function is useful for calculating integrals of time-series data, like that generated by the *fix ave/correlate* command.

The slope() function uses linear regression to fit a line to the set of points, equally spaced by 1 in their x coordinate: (1,V1), (2,V2), ..., (N,VN), where the Vi are the values in the global vector of length N. The returned value is the slope of the line. If the line has a single point or is vertical, it returns 1.0e20.

Added in version 27June2024.

The sort(x) and rsort(x) functions sort the data of the input vector by their numeric value: sort(x) sorts in ascending order, rsort(x) sorts in descending order.

The gmask(x) function takes 1 argument which is a group ID. It can only be used in atom-style variables. It returns a 1 for atoms that are in the group, and a 0 for atoms that are not.

The rmask(x) function takes 1 argument which is a region ID. It can only be used in atom-style variables. It returns a 1 for atoms that are in the geometric region, and a 0 for atoms that are not.

The grmask(x,y) function takes 2 arguments. The first is a group ID, and the second is a region ID. It can only be used in atom-style variables. It returns a 1 for atoms that are in both the group and region, and a 0 for atoms that are not in both.

The next(x) function takes 1 argument which is a variable ID (not "v_foo", just "foo"). It must be for a file-style or atomfile-style variable. Each time the next() function is invoked (i.e. each time the equal-style or atom-style variable is evaluated), the following steps occur.

For file-style variables, the current string value stored by the file-style variable is converted to a numeric value and returned by the function. And the next string value in the file is read and stored. Note that if the line previously read from the file was not a numeric string, then it will typically evaluate to 0.0, which is likely not what you want.

For atomfile-style variables, the current per-atom values stored by the atomfile-style variable are returned by the function. And the next set of per-atom values in the file is read and stored.

Since file-style and atomfile-style variables read and store the first line of the file or first set of per-atoms values when they are defined in the input script, these are the value(s) that will be returned the first time the next() function is invoked. If next() is invoked more times than there are lines or sets of lines in the file, the variable is deleted, similar to how the *next* command operates.

The is_file(name) function is a test whether *name* is a (readable) file and returns 1 in this case, otherwise it returns 0. For that *name* is taken as a literal string and must not have any blanks in it.

The is_os(name) function is a test whether *name* is part of the OS information that LAMMPS collects and provides in the platform::os_info() function. The argument *name* is interpreted as a regular expression as documented for the utils::strmatch() function. This allows to adapt LAMMPS inputs to the OS it runs on:

```
if $(is_os(^Windows)) then &
  "shell copy ${input_dir}\some_file.txt ." &
else &
  "shell cp ${input_dir}/some_file.txt ."
```

The extract_setting(name) function enables access to basic settings for the LAMMPS executable and the running simulation via calling the lammps_extract_setting() library function. For example, the number of processors (MPI ranks) being used by the simulation or the MPI process ID (for this processor) can be queried, or the number of atom types, bond types and so on. For the full list of available keywords *name* and their meaning, see the documentation for extract_setting() via the link in this paragraph.

The label2type(kind,label) function converts type labels into numeric types, using label maps created by the *labelmap* or *read_data* commands. The first argument is the label map kind (atom, bond, angle, dihedral, or improper) and the second argument is the label. The function returns the corresponding numeric type or triggers an error if the queried label does not exist.

Added in version 15Jun2023.

The is_typelabel(kind,label) function has the same arguments as label2type(), but returns 1 if the type label has been assigned, otherwise it returns 0. This function can be used to check if a particular type label already exists in the simulation.

Added in version 29Aug2024.

The is_timeout() function returns 1 when the *timer timeout* has expired otherwise it returns 0. This function can be used to check inputs in combination with the *if command* to execute commands after the timer has expired. Example:

```
variable timeout equal is_timeout()
timer timeout 0:10:00 every 10
run 10000
if ${timeout} then "print 'Timer has expired'"
```

Feature Functions

Feature functions allow probing of the running LAMMPS executable for whether specific features are available, active, or defined. All 3 of the functions take two arguments, a *category* and a category-specific second argument. Both are strings and thus cannot be formulas themselves; only $-style immediate variable expansion is possible. The return value of the functions is either 1.0 or 0.0 depending on whether the function evaluates to true or false, respectively.

The *is_available(category,name)* function queries whether a specific feature is available in the LAMMPS executable that is being run, i.e whether it was included or enabled at compile time.

This supports the following categories: *command*, *compute*, *fix*, *pair_style* and *feature*. For all the categories except *feature* the *name* is a style name, e.g. *nve* for the *fix* category. Note that many LAMMPS input script commands such as *create_atoms* are actually instances of a command style which LAMMPS defines, as opposed to built-in commands. For all of these styles except *command*, appending of active suffixes is also tried before reporting failure.

The *feature* category checks the availability of the following compile-time enabled features: GZIP support, PNG support, JPEG support, FFMPEG support, and C++ exceptions for error handling. Corresponding names are *gzip*, *png*, *jpeg*, *ffmpeg* and *exceptions*.

Example: Only dump in a given format if the compiled binary supports it.

```
if "$(is_available(feature,png))" then "print 'PNG supported'" else "print 'PNG not supported'"
if "$(is_available(feature,ffmpeg)" then "dump 3 all movie 25 movie.mp4 type type zoom 1.6 adiam 1.0"
```

The *is_active(category,feature)* function queries whether a specific feature is currently active within LAMMPS. The features are grouped by categories. Supported categories and features are:

- *package*: features = *gpu* or *intel* or *kokkos* or *omp*
- *newton*: features = *pair* or *bond* or *any*

- *pair*: features = *single* or *respa* or *manybody* or *tail* or *shift*
- *comm_style*: features = *brick* or *tiled*
- *min_style*: features = a minimizer style name
- *run_style*: features = a run style name
- *atom_style*: features = an atom style name
- *pair_style*: features = a pair style name
- *bond_style*: features = a bond style name
- *angle_style*: features = an angle style name
- *dihedral_style*: features = a dihedral style name
- *improper_style*: features = an improper style name
- *kspace_style*: features = a kspace style name

Most of the settings are self-explanatory. For the *package* category, a package may have been included in the LAMMPS build, but not have enabled by any input script command, and hence be inactive. The *single* feature in the *pair* category checks whether the currently defined pair style supports a Pair::single() function as needed by compute group/group and others features or LAMMPS. Similarly, the *respa* feature checks whether the inner/middle/outer mode of r-RESPA is supported by the current pair style.

For the categories with *style* in their name, only a single instance of the style is ever active at any time in a LAMMPS simulation. Thus the check is whether the currently active style matches the specified name. This check is also done using suffix flags, if available and enabled.

Example 1: Disable use of suffix for PPPM when using GPU package (i.e. run it on the CPU concurrently while running the pair style on the GPU), but do use the suffix otherwise (e.g. with OPENMP).

```
pair_style lj/cut/coul/long 14.0
if $(is_active(package,gpu)) then "suffix off"
kspace_style pppm
```

Example 2: Use r-RESPA with inner/outer cutoff, if supported by the current pair style, otherwise fall back to using r-RESPA with simply the pair keyword and reducing the outer time step.

```
timestep $(2.0*(1.0+2.0*is_active(pair,respa)))
if $(is_active(pair,respa)) then "run_style respa 4 3 2 2 improper 1 inner 2 5.5 7.0 outer 3 kspace 4" else
 ↪"run_style respa 3 3 2 improper 1 pair 2 kspace 3"
```

The *is_defined(category,id)* function checks whether an instance of a style or variable with a specific ID or name is currently defined within LAMMPS. The supported categories are *compute*, *dump*, *fix*, *group*, *region*, and *variable*. Each of these styles (as well as the variable command) can be specified multiple times within LAMMPS, each with a unique *id*. This function checks whether the specified *id* exists. For category *variable", the *id* is the variable name.

Atom Values and Vectors

Atom values take an integer argument I from 1 to N, where I is the atom-ID, e.g. x[243], which means use the x coordinate of the atom with ID = 243. Or they can take a variable name, specified as v_name, where name is the name of the variable, like x[v_myIndex]. The variable can be of any style except *vector* or *atom* or *atomfile* variables. The variable is evaluated and the result is expected to be numeric and is cast to an integer (i.e. 3.4 becomes 3), to use an index, which must be a value from 1 to N. Note that a "formula" cannot be used as the argument between the brackets, e.g. x[243+10] or x[v_myIndex+1] are not allowed. To do this a single variable can be defined that contains the needed formula.

Note that the 0 < atom-ID <= N, where N is the largest atom ID in the system. If an ID is specified for an atom that does not currently exist, then the generated value is 0.0.

Atom vectors generate one value per atom, so that a reference like "vx" means the x-component of each atom's velocity will be used when evaluating the variable.

The meaning of the different atom values and vectors is mostly self-explanatory. *Mol* refers to the molecule ID of an atom, and is only defined if an *atom_style* is being used that defines molecule IDs.

Note that many other atom attributes can be used as inputs to a variable by using the *compute property/atom* command and then referencing that compute.

Custom atom properties

Added in version 7Feb2024.

Custom atom properties refer to per-atom integer and floating point vectors or arrays that have been added via the *fix property/atom* command. When that command is used specific names are given to each attribute which are the "name" portion of these references. References beginning with *i* and *d* refer to integer and floating point properties respectively. Per-atom vectors are referenced by *i_name* and *d_name*; per-atom arrays are referenced by *i2_name* and *d2_name*.

The various allowed references to integer custom atom properties in the variable formulas for equal-, vector-, and atom-style variables are listed in the following table. References to floating point custom atom properties are the same; just replace the leading "i" with "d".

equal	i_name[I]	element of per-atom vector (I = atom ID)
equal	i2_name[I][J]	element of per-atom array (I = atom ID)
vector	i_name[I]	element of per-atom vector (I = atom ID)
vector	i2_name[I][J]	element of per-atom array (I = atom ID)
atom	i_name	per-atom vector
atom	i2_name[I]	column of per-atom array

The I and J indices in these custom atom property references can be integers or can be a variable name, specified as v_name, where name is the name of the variable. The rules for this syntax are the same as for indices in the "Atom Values and Vectors" discussion above.

Compute References

Compute references access quantities calculated by a *compute*. The ID in the reference should be replaced by the ID of a compute defined elsewhere in the input script.

As discussed on the page for the *compute* command, computes can produce global, per-atom, local, and per-grid values. Only global and per-atom values can be used in a variable. Computes can also produce scalars (global only), vectors, and arrays. See the doc pages for individual computes to see what different kinds of data they produce.

An equal-style variable can only use scalar values, either from global or per-atom data. In the case of per-atom data, this would be a value for a specific atom.

A vector-style variable can use scalar values (same as for equal-style variables), or global vectors of values. The latter can also be a column of a global array.

Atom-style variables can use scalar values (same as for equal-style variables), or per-atom vectors of values. The latter can also be a column of a per-atom array.

The various allowed compute references in the variable formulas for equal-, vector-, and atom-style variables are listed in the following table:

equal	c_ID	global scalar
equal	c_ID[I]	element of global vector
equal	c_ID[I][J]	element of global array
equal	C_ID[I]	element of per-atom vector (I = atom ID)
equal	C_ID[I][J]	element of per-atom array (I = atom ID)
vector	c_ID	global vector
vector	c_ID[I]	column of global array
atom	c_ID	per-atom vector
atom	c_ID[I]	column of per-atom array

Note that if an equal-style variable formula wishes to access per-atom data from a compute, it must use capital "C" as the ID prefix and not lower-case "c".

Also note that if a vector- or atom-style variable formula needs to access a scalar value from a compute (i.e. the 5 kinds of values in the first 5 lines of the table), it can not do so directly. Instead, it can use a reference to an equal-style variable which stores the scalar value from the compute.

The I and J indices in these compute references can be integers or can be a variable name, specified as v_name, where name is the name of the variable. The rules for this syntax are the same as for indices in the "Atom Values and Vectors" discussion above.

If a variable containing a compute is evaluated directly in an input script (not during a run), then the values accessed by the compute should be current. See the discussion below about "Variable Accuracy".

Fix References

Fix references access quantities calculated by a *fix*. The ID in the reference should be replaced by the ID of a fix defined elsewhere in the input script.

As discussed on the page for the *fix* command, fixes can produce global, per-atom, local, and per-grid values. Only global and per-atom values can be used in a variable. Fixes can also produce scalars (global only), vectors, and arrays. See the doc pages for individual fixes to see what different kinds of data they produce.

An equal-style variable can only use scalar values, either from global or per-atom data. In the case of per-atom data, this would be a value for a specific atom.

A vector-style variable can use scalar values (same as for equal-style variables), or global vectors of values. The latter can also be a column of a global array.

Atom-style variables can use scalar values (same as for equal-style variables), or per-atom vectors of values. The latter can also be a column of a per-atom array.

The allowed fix references in variable formulas for equal-, vector-, and atom-style variables are listed in the following table:

equal	f_ID	global scalar
equal	f_ID[I]	element of global vector
equal	f_ID[I][J]	element of global array
equal	F_ID[I]	element of per-atom vector (I = atom ID)
equal	F_ID[I][J]	element of per-atom array (I = atom ID)
vector	f_ID	global vector
vector	f_ID[I]	column of global array
atom	f_ID	per-atom vector
atom	f_ID[I]	column of per-atom array

Note that if an equal-style variable formula wishes to access per-atom data from a fix, it must use capital "F" as the ID prefix and not lower-case "f".

Also note that if a vector- or atom-style variable formula needs to access a scalar value from a fix (i.e. the 5 kinds of values in the first 5 lines of the table), it can not do so directly. Instead, it can use a reference to an equal-style variable which stores the scalar value from the fix.

The I and J indices in these fix references can be integers or can be a variable name, specified as v_name, where name is the name of the variable. The rules for this syntax are the same as for indices in the "Atom Values and Vectors" discussion above.

Note that some fixes only generate quantities on certain timesteps. If a variable attempts to access the fix on non-allowed timesteps, an error is generated. For example, the *fix ave/time* command may only generate averaged quantities every 100 steps. See the doc pages for individual fix commands for details.

If a variable containing a fix is evaluated directly in an input script (not during a run), then the values accessed by the fix should be current. See the discussion below about "Variable Accuracy".

Variable References

Variable references access quantities stored or calculated by other variables, which will cause those variables to be evaluated. The name in the reference should be replaced by the name of a variable defined elsewhere in the input script.

As discussed on this doc page, equal-style variables generate a single global numeric value, vector-style variables generate a vector of global numeric values, and atom-style and atomfile-style variables generate a per-atom vector of numeric values. All other variables store one or more strings.

The formula for an equal-style variable can use any style of variable including a vector_style or atom-style or atomfile-style. For these 3 styles, a subscript must be used to access a single value from the vector-, atom-, or atomfile-style variable. If a string-storing variable is used, the string is converted to a numeric value. Note that this will typically produce a 0.0 if the string is not a numeric string, which is likely not what you want.

The formula for a vector-style variable can use any style of variable, including atom-style or atomfile-style variables. For these 2 styles, a subscript must be used to access a single value from the atom-, or atomfile-style variable.

The formula for an atom-style variable can use any style of variable, including other atom-style or atomfile-style variables. If it uses a vector-style variable, a subscript must be used to access a single value from the vector-style variable.

The allowed variable references in variable formulas for equal-, vector-, and atom-style variables are listed in the following table. Note that there is no ambiguity as to what a reference means, since referenced variables produce only a global scalar or global vector or per-atom vector.

equal	v_name	global scalar from an equal-style variable
equal	v_name[I]	element of global vector from a vector-style variable
equal	v_name[I]	element of per-atom vector (I = atom ID) from an atom- or atomfile-style variable
vector	v_name	global scalar from an equal-style variable
vector	v_name	global vector from a vector-style variable
vector	v_name[I]	element of global vector from a vector-style variable
vector	v_name[I]	element of per-atom vector (I = atom ID) from an atom- or atomfile-style variable
atom	v_name	global scalar from an equal-style variable
atom	v_name	per-atom vector from an atom-style or atomfile-style variable
atom	v_name[I]	element of global vector from a vector-style variable
atom	v_name[I]	element of per-atom vector (I = atom ID) from an atom- or atomfile-style variable

For the I index, an integer can be specified or a variable name, specified as v_name, where name is the name of the variable. The rules for this syntax are the same as for indices in the "Atom Values and Vectors" discussion above.

Vector Initialization

Added in version 15Jun2023.

Vector-style variables only can be initialized with a special syntax, instead of using a formula. The syntax is a bracketed, comma-separated syntax like the following:

```
variable myvec vector [1,3.5,7,10.2]
```

The 3rd argument formula is replaced by the vector values in brackets, separated by commas. This example creates a 4-length vector with specific numeric values, each of which can be specified as an integer or floating point value.

Note that while whitespace can be added before or after individual values, no other mathematical operations can be specified. E.g. "3*10" or "3*v_abc" are not valid vector elements, nor is "10*[1,2,3,4]" valid for the entire vector.

Unlike vector variables specified with formulas, this vector variable is static; its length and values never changes. Its values can be used in other commands (including vector-style variables specified with formulas) via the usual syntax for accessing individual vector elements or the entire vector.

1.114.4 Immediate Evaluation of Variables

If you want an equal-style variable to be evaluated immediately, it may be the case that you do not need to define a variable at all. See the *Commands parse* page for info on how to use "immediate" variables in an input script, specified as $(formula) with parenthesis, where the formula has the same syntax as equal-style variables described on this page. This effectively evaluates a formula immediately without using the variable command to define a named variable.

More generally, there is a difference between referencing a variable with a leading $ sign (e.g. $x or ${abc}) versus with a leading "v_" (e.g. v_x or v_abc). The former can be used in any input script command, including a variable command. The input script parser evaluates the reference variable immediately and substitutes its value into the command. As explained on the *Commands parse* doc page, you can also use un-named "immediate" variables for this purpose. For example, a string like this $((xlo+xhi)/2+sqrt(v_area)) in an input script command evaluates the string between the parenthesis as an equal-style variable formula.

Referencing a variable with a leading "v_" is an optional or required kind of argument for some commands (e.g. the *fix ave/chunk* or *dump custom* or *thermo_style* commands) if you wish it to evaluate a variable periodically during a run. It can also be used in a variable formula if you wish to reference a second variable. The second variable will be evaluated whenever the first variable is evaluated.

As an example, suppose you use this command in your input script to define the variable "v" as

```
variable v equal vol
```

before a run where the simulation box size changes. You might think this will assign the initial volume to the variable "v". That is not the case. Rather it assigns a formula which evaluates the volume (using the thermo_style keyword "vol") to the variable "v". If you use the variable "v" in some other command like *fix ave/time* then the current volume of the box will be evaluated continuously during the run.

If you want to store the initial volume of the system, you can do it this way:

```
variable v equal vol
variable v0 equal $v
```

The second command will force "v" to be evaluated (yielding the initial volume) and assign that value to the variable "v0". Thus the command

```
thermo_style custom step v_v v_v0
```

would print out both the current and initial volume periodically during the run.

Note that it is a mistake to enclose a variable formula in double quotes if it contains variables preceded by $ signs. For example,

```
variable vratio equal "${vfinal}/${v0}"
```

This is because the quotes prevent variable substitution (explained on the *Commands parse* doc page), and thus an error will occur when the formula for "vratio" is evaluated later.

1.114.5 Variable Accuracy

Obviously, LAMMPS attempts to evaluate variables which contain formulas (*equal* and *vector* and *atom* style variables) accurately whenever the evaluation is performed. Depending on what is included in the formula, this may require invoking a *compute*, either directly or indirectly via a thermo keyword, or accessing a value previously calculated by a compute, or accessing a value calculated and stored by a *fix*. If the compute is one that calculates the energy or pressure of the system, then the corresponding energy or virial quantities need to be tallied during the evaluation of the interatomic potentials (pair, bond, etc) on any timestep that the variable needs the tallies. An input script can also request variables be evaluated before or after or in between runs, e.g. by including them in a *print* command.

LAMMPS keeps track of all of this as it performs a *run* or *minimize* simulation, as well as in between simulations. An error will be generated if you attempt to evaluate a variable when LAMMPS knows it cannot produce accurate values. For example, if a *thermo_style custom* command prints a variable which accesses values stored by a *fix ave/time* command and the timesteps on which thermo output is generated are not multiples of the averaging frequency used in the fix command, then an error will occur.

However, there are two special cases to be aware when a variable requires invocation of a compute (directly or indirectly). The first is if the variable is evaluated before the first *run* or *minimize* command in the input script. In this case, LAMMPS will generate an error. This is because many computes require initializations which have not yet taken place. One example is the calculation of degrees of freedom for temperature computes. Another example are the computes mentioned above which require tallying of energy or virial quantities; these values are not tallied until the first simulation begins.

The second special case is when a variable that depends on a compute is evaluated in between *run* or *minimize* commands. It is possible for other input script commands issued following the previous run, but before the variable is evaluated, to change the system. For example, the *delete_atoms* command could be used to remove atoms. Since the compute will not re-initialize itself until the next simulation or it may depend on energy/virial computations performed before the system was changed, it will potentially generate an incorrect answer when evaluated. Note that LAMMPS will not generate an error in this case; the evaluated variable may simply be incorrect.

The way to get around both of these special cases is to perform a 0-timestep run before evaluating the variable. For example, these commands

```
# delete_atoms random fraction 0.5 yes all NULL 49839
# run 0 post no
variable t equal temp    # this thermo keyword invokes a temperature compute
print "Temperature of system = $t"
run 1000
```

will generate an error if the "run 1000" command is the first simulation in the input script. If there were a previous run, these commands will print the correct temperature of the system. But if the *delete_atoms* command is uncommented, the printed temperature will be incorrect, because information stored by temperature compute is no longer valid.

Both these issues are resolved, if the "run 0" command is uncommented. This is because the "run 0" simulation will initialize (or re-initialize) the temperature compute correctly.

1.114.6 Restrictions

Indexing any formula element by global atom ID, such as an atom value, requires the *atom style* to use a global mapping in order to look up the vector indices. By default, only atom styles with molecular information create global maps. The *atom_modify map* command can override the default, e.g. for atomic-style atom styles.

All *universe*- and *uloop*-style variables defined in an input script must have the same number of values.

1.114.7 Related commands

next, *jump*, *include*, *temper*, *fix print*, *print*

1.114.8 Default

none

1.115 velocity command

1.115.1 Syntax

velocity group-ID style args keyword value ...

- group-ID = ID of group of atoms whose velocity will be changed
- style = *create* or *set* or *scale* or *ramp* or *zero*

  create args = temp seed
    temp = temperature value (temperature units)
    seed = random # seed (positive integer)
  set args = vx vy vz
    vx,vy,vz = velocity value or NULL (velocity units)
    any of vx,vy,vz can be a variable (see below)
  scale arg = temp
    temp = temperature value (temperature units)
  ramp args = vdim vlo vhi dim clo chi
    vdim = vx or vy or vz
    vlo,vhi = lower and upper velocity value (velocity units)
    dim = x or y or z
    clo,chi = lower and upper coordinate bound (distance units)
  zero arg = linear or angular
    linear = zero the linear momentum
    angular = zero the angular momentum

- zero or more keyword/value pairs may be appended
- keyword = *dist* or *sum* or *mom* or *rot* or *temp* or *bias* or *loop* or *rigid* or *units*

  dist value = uniform or gaussian
  sum value = no or yes
  mom value = no or yes
  rot value = no or yes
  temp value = temperature compute ID
  bias value = no or yes

loop value = all or local or geom
rigid value = fix-ID
  fix-ID = ID of rigid body fix
units value = box or lattice

1.115.2 Examples

```
velocity all create 300.0 4928459 rot yes dist gaussian
velocity border set NULL 4.0 v_vz sum yes units box
velocity flow scale 300.0
velocity flow ramp vx 0.0 5.0 y 5 25 temp mytemp
velocity all zero linear
```

1.115.3 Description

Set or change the velocities of a group of atoms in one of several styles. For each style, there are required arguments and optional keyword/value parameters. Not all options are used by each style. Each option has a default as listed below.

The *create* style generates an ensemble of velocities using a random number generator with the specified seed at the specified temperature.

The *set* style sets the velocities of all atoms in the group to the specified values. If any component is specified as NULL, then it is not set. Any of the vx,vy,vz velocity components can be specified as an equal-style or atom-style *variable*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated, and its value used to determine the velocity component. Note that if a variable is used, the velocity it calculates must be in box units, not lattice units; see the discussion of the *units* keyword below.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters or other parameters.

Atom-style variables can specify the same formulas as equal-style variables but can also include per-atom values, such as atom coordinates. Thus it is easy to specify a spatially-dependent velocity field.

The *scale* style computes the current temperature of the group of atoms and then rescales the velocities to the specified temperature.

The *ramp* style is similar to that used by the *compute temp/ramp* command. Velocities ramped uniformly from vlo to vhi are applied to dimension vx, or vy, or vz. The value assigned to a particular atom depends on its relative coordinate value (in dim) from clo to chi. For the example above, an atom with y-coordinate of 10 (1/4 of the way from 5 to 25), would be assigned a x-velocity of 1.25 (1/4 of the way from 0.0 to 5.0). Atoms outside the coordinate bounds (less than 5 or greater than 25 in this case), are assigned velocities equal to vlo or vhi (0.0 or 5.0 in this case).

The *zero* style adjusts the velocities of the group of atoms so that the aggregate linear or angular momentum is zero. No other changes are made to the velocities of the atoms. If the *rigid* option is specified (see below), then the zeroing is performed on individual rigid bodies, as defined by the *fix rigid or fix rigid/small* commands. In other words, zero linear will set the linear momentum of each rigid body to zero, and zero angular will set the angular momentum of each rigid body to zero. This is done by adjusting the velocities of the atoms in each rigid body.

All temperatures specified in the velocity command are in temperature units; see the *units* command. The units of velocities and coordinates depend on whether the *units* keyword is set to *box* or *lattice*, as discussed below.

For all styles, no atoms are assigned z-component velocities if the simulation is 2d; see the *dimension* command.

The keyword/value options are used in the following ways by the various styles.

The *dist* keyword is used by *create*. The ensemble of generated velocities can be a *uniform* distribution from some minimum to maximum value, scaled to produce the requested temperature. Or it can be a *gaussian* distribution with a mean of 0.0 and a sigma scaled to produce the requested temperature.

The *sum* keyword is used by all styles, except *zero*. The new velocities will be added to the existing ones if sum = yes, or will replace them if sum = no.

The *mom* and *rot* keywords are used by *create*. If mom = yes, the linear momentum of the newly created ensemble of velocities is zeroed; if rot = yes, the angular momentum is zeroed.

If specified, the *temp* keyword is used by *create* and *scale* to specify a *compute* that calculates temperature in a desired way, e.g. by first subtracting out a velocity bias, as discussed on the *Howto thermostat* doc page. If this keyword is not specified, *create* and *scale* calculate temperature using a compute that is defined internally as follows:

```
compute velocity_temp group-ID temp
```

where group-ID is the same ID used in the velocity command. i.e. the group of atoms whose velocity is being altered. This compute is deleted when the velocity command is finished. See the *compute temp* command for details. If the calculated temperature should have degrees-of-freedom removed due to fix constraints (e.g. SHAKE or rigid-body constraints), then the appropriate fix command must be specified before the velocity command is issued.

The *bias* keyword with a *yes* setting is used by *create* and *scale*, but only if the *temp* keyword is also used to specify a *compute* that calculates temperature in a desired way. If the temperature compute also calculates a velocity bias, the bias is subtracted from atom velocities before the *create* and *scale* operations are performed. After the operations, the bias is added back to the atom velocities. See the *Howto thermostat* page for more discussion of temperature computes with biases. Note that the velocity bias is only applied to atoms in the temperature compute specified with the *temp* keyword.

As an example, assume atoms are currently streaming in a flow direction (which could be separately initialized with the *ramp* style), and you wish to initialize their thermal velocity to a desired temperature. In this context thermal velocity means the per-particle velocity that remains when the streaming velocity is subtracted. This can be done using the *create* style with the *temp* keyword specifying the ID of a *compute temp/ramp* or *compute temp/profile* command, and the *bias* keyword set to a *yes* value.

The *loop* keyword is used by *create* in the following ways.

If loop = all, then each processor loops over all atoms in the simulation to create velocities, but only stores velocities for atoms it owns. This can be a slow loop for a large simulation. If atoms were read from a data file, the velocity assigned to a particular atom will be the same, independent of how many processors are being used. This will not be the case if atoms were created using the *create_atoms* command, since atom IDs will likely be assigned to atoms differently.

If loop = local, then each processor loops over only its atoms to produce velocities. The random number seed is adjusted to give a different set of velocities on each processor. This is a fast loop, but the velocity assigned to a particular atom will depend on which processor owns it. Thus the results will always be different when a simulation is run on a different number of processors.

If loop = geom, then each processor loops over only its atoms. For each atom a unique random number seed is created, based on the atom's xyz coordinates. A velocity is generated using that seed. This is a fast loop and the velocity assigned to a particular atom will be the same, independent of how many processors are used. However, the set of generated velocities may be more correlated than if the *all* or *local* keywords are used.

Note that the *loop geom* keyword will not necessarily assign identical velocities for two simulations run on different machines. This is because the computations based on xyz coordinates are sensitive to tiny differences in the double-precision value for a coordinate as stored on a particular machine.

The *rigid* keyword only has meaning when used with the *zero* style. It allows specification of a fix-ID for one of the *rigid-body fix* variants which defines a set of rigid bodies. The zeroing of linear or angular momentum is then performed for each rigid body defined by the fix, as described above.

The *units* keyword is used by *set* and *ramp*. If units = box, the velocities and coordinates specified in the velocity command are in the standard units described by the *units* command (e.g. Angstroms/fs for real units). If units = lattice, velocities are in units of lattice spacings per time (e.g. spacings/fs) and coordinates are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacing.

1.115.4 Restrictions

Assigning a temperature via the *create* style to a system with *rigid bodies* or *SHAKE constraints* may not have the desired outcome for two reasons. First, the velocity command can be invoked before all of the relevant fixes are created and initialized and the number of adjusted degrees of freedom (DOFs) is known. Thus it is not possible to compute the target temperature correctly. Second, the assigned velocities may be partially canceled when constraints are first enforced, leading to a different temperature than desired. A workaround for this is to perform a *run 0* command, which ensures all DOFs are accounted for properly, and then rescale the temperature to the desired value before performing a simulation. For example:

```
velocity all create 300.0 12345
run 0                          # temperature may not be 300K
velocity all scale 300.0         # now it should be
```

1.115.5 Related commands

fix rigid, *fix shake*, *lattice*

1.115.6 Default

The keyword defaults are dist = uniform, sum = no, mom = yes, rot = no, bias = no, loop = all, and units = lattice. The temp and rigid keywords are not defined by default.

1.116 write_coeff command

1.116.1 Syntax

```
write_coeff file
```

- file = name of data file to write out

1.116.2 Examples

```
write_coeff polymer.coeff
```

1.116.3 Description

Write a text format file with the currently defined force field coefficients in a way, that it can be read by LAMMPS with the *include* command. In combination with the nocoeff option of *write_data* this can be used to move the Coeffs sections from a data file into a separate file.

Note

The write_coeff command is not yet fully implemented as some pair styles do not output their coefficient information. This means you will need to add/copy this information manually.

1.116.4 Restrictions

none

1.116.5 Related commands

read_data, *write_restart*, *write_data*

1.117 write_data command

1.117.1 Syntax

```
write_data file keyword value ...
```

- file = name of data file to write out
- zero or more keyword/value pairs may be appended
- keyword = *nocoeff* or *nofix* or *nolabelmap* or *triclinic/general* or *types* or *pair*

  nocoeff = do not write out force field info
  nofix = do not write out extra sections read by fixes
  nolabelmap = do not write out type labels
  triclinic/general = write data file in general triclinic format
  types value = numeric or labels
  pair value = ii or ij
    ii = write one line of pair coefficient info per atom type
    ij = write one line of pair coefficient info per IJ atom type pair

1.117.2 Examples

```
write_data data.polymer
write_data data.*
write_data data.solid triclinic/general
```

1.117.3 Description

Write a data file in text format of the current state of the simulation. Data files can be read by the *read data* command to begin a simulation. The *read_data* command also describes their format.

Similar to *dump* files, the data filename can contain a "*" wild-card character. The "*" is replaced with the current timestep value.

Data in Coeff sections

The write_data command may not always write all coefficient settings to the corresponding Coeff sections of the data file. This can have one of multiple reasons. 1) The style may be a hybrid style. In that case *no* coeff information is written. 2) A few styles may be missing the code that would write those sections (This is rare these days, but if you come across one, please notify the LAMMPS developers). 3) Some pair styles require a single pair_coeff statement and those are not compatible with data files. 4) The default for write_data is to write a PairCoeff section, which has only entries for atom types i == j. The remaining coefficients would be inferred through the currently selected mixing rule. If there has been a pair_coeff command with i != j, this setting would be lost. LAMMPS will detect this and print a warning message unless *pair ij* is appended to the write_data command. This will request writing a PairIJCoeff section which has information for all pairs of atom types. In cases where the coefficient data in the data file is incomplete, you will need to re-specify that information in your input script that reads the data file.

Because a data file is in text format, if you use a data file written out by this command to restart a simulation, the initial state of the new run will be slightly different than the final state of the old run (when the file was written) which was represented internally by LAMMPS in binary format. A new simulation which reads the data file will thus typically diverge from a simulation that continued in the original input script.

If you want to do more exact restarts, using binary files, see the *restart*, *write_restart*, and *read_restart* commands. You can also convert binary restart files to text data files, after a simulation has run, using the *-r command-line switch*.

Note

Only limited information about a simulation is stored in a data file. For example, no information about atom *groups* and *fixes* are stored. *Binary restart files* store more information.

Bond interactions (angle, etc) that have been turned off by the *fix shake* or *delete_bonds* command will be written to a data file as if they are turned on. This means they will need to be turned off again in a new run after the data file is read.

Bonds that are broken (e.g. by a bond-breaking potential) are not written to the data file. Thus these bonds will not exist when the data file is read.

Use of the *nocoeff* keyword means no force field parameters are written to the data file. This can be helpful, for example, if you want to make significant changes to the force field or if the force field parameters are read in separately, e.g. from an include file.

Use of the *nofix* keyword means no extra sections read by fixes are written to the data file (see the *fix* option of the *read_data* command for details). For example, this option excludes sections for user-created per-atom properties from *fix property/atom*.

The *nolabelmap* and *types* keywords refer to type labels that may be defined for numeric atom types, bond types, angle types, etc. The label map can be defined in two ways, either by the *labelmap* command or in data files read by the *read_data* command which have sections for Atom Type Labels, Bond Type Labels, Angle Type Labels, etc. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

Use of the *nolabelmap* keyword means that even if type labels exist for a given type-kind (Atoms, Bonds, Angles, etc.), type labels are not written to the data file. By default, they are written if they exist. A type label must be defined for every numeric type (within a given type-kind) to be written to the data file.

Use of the *triclinic/general* keyword will output a data file which specifies a general triclinic simulation box as well as per-atom quantities consistent with the general triclinic box. The latter means that per-atom vectors, such as velocities and dipole moments will be oriented consistent with the 3d rotation implied by the general triclinic box (relative to the associated restricted triclinic box).

This option can only be requested if the simulation box was initially defined to be general triclinic. If if was and the *triclinic/general* keyword is not used, then the data file will specify a restricted triclinic box, since that is the internal format LAMMPS uses for both general and restricted triclinic simulations. See the *Howto triclinic* doc page for more explanation of how general triclinic simulation boxes are supported by LAMMPS. And see the *read_data* doc page for details of how the format is altered for general triclinic data files.

The *types* keyword determines how atom types, bond types, angle types, etc are written into these data file sections: Atoms, Bonds, Angles, etc. The default is the *numeric* setting, even if type label maps exist. If the *labels* setting is used, type labels will be written to the data file, if the corresponding label map exists. Note that when using *types labels*, the *nolabelmap* keyword cannot be used.

The *pair* keyword lets you specify in what format the pair coefficient information is written into the data file. If the value is specified as *ii*, then one line per atom type is written, to specify the coefficients for each of the I=J interactions. This means that no cross-interactions for I != J will be specified in the data file and the pair style will apply its mixing rule, as documented on individual *pair_style* doc pages. Of course this behavior can be overridden in the input script after reading the data file, by specifying additional *pair_coeff* commands for any desired I,J pairs.

If the value is specified as *ij*, then one line of coefficients is written for all I,J pairs where I <= J. These coefficients will include any specific settings made in the input script up to that point. The presence of these I != J coefficients in the data file will effectively turn off the default mixing rule for the pair style. Again, the coefficient values in the data file can be overridden in the input script after reading the data file, by specifying additional *pair_coeff* commands for any desired I,J pairs.

1.117.4 Restrictions

This command requires inter-processor communication to migrate atoms before the data file is written. This means that your system must be ready to perform a simulation before using this command (force fields setup, atom masses initialized, etc).

1.117.5 Related commands

read_data, *write_restart*

1.117.6 Default

The option defaults are pair = ii and types = numeric.

1.118 write_dump command

1.118.1 Syntax

```
write_dump group-ID style file dump-args modify dump_modify-args
```

- group-ID = ID of the group of atoms to be dumped
- style = any of the supported *dump styles*
- file = name of file to write dump info to
- dump-args = any additional args needed for a particular *dump style*
- modify = all args after this keyword are passed to *dump_modify* (optional)
- dump-modify-args = args for *dump_modify* (optional)

1.118.2 Examples

```
write_dump all atom dump.atom
write_dump subgroup atom dump.run.bin
write_dump all custom dump.myforce.* id type x y vx fx
write_dump flow custom dump.%.myforce id type c_myF[3] v_ke modify sort id
write_dump all xyz system.xyz modify sort id element O H
write_dump all image snap*.jpg type type size 960 960 modify backcolor white
write_dump all image snap*.jpg element element &
  bond atom 0.3 shiny 0.1 ssao yes 6345 0.2 size 1600 1600  &
  modify backcolor white element C C O H N C C C O H H S O H
write_dump all atom/gz dump.atom.gz modify compression_level 9
write_dump flow custom/zstd dump.%.myforce.zst &
             id type c_myF[3] v_ke &
             modify sort id &
             compression_level 15
```

1.118.3 Description

Dump a single snapshot of atom quantities to one or more files for the current state of the system. This is a one-time immediate operation, in contrast to the *dump* command which will will set up a dump style to write out snapshots periodically during a running simulation.

The syntax for this command is mostly identical to that of the *dump* and *dump_modify* commands as if they were concatenated together, with the following exceptions: There is no need for a dump ID or dump frequency and the keyword *modify* is added. The latter is so that the full range of *dump_modify* options can be specified for the single snapshot, just as they can be for multiple snapshots. The *modify* keyword separates the arguments that would normally be passed to the *dump* command from those that would be given the *dump_modify*. Both support optional arguments and thus LAMMPS needs to be able to cleanly separate the two sets of args.

Note that if the specified filename uses wildcard characters "*" or "%", as supported by the *dump* command, they will operate in the same fashion to create the new filename(s). Normally, *dump image* files require a filename with a "*" character for the timestep. That is not the case for the write_dump command; no wildcard "*" character is necessary.

1.118.4 Restrictions

All restrictions for the *dump* and *dump_modify* commands apply to this command as well, with the exception of the *dump image* filename not requiring a wildcard "*" character, as noted above.

Since dumps are normally written during a *run* or *energy minimization*, the simulation has to be ready to run before this command can be used. Similarly, if the dump requires information from a compute, fix, or variable, the information needs to have been calculated for the current timestep (e.g. by a prior run), else LAMMPS will generate an error message.

For example, it is not possible to dump per-atom energy with this command before a run has been performed, since no energies and forces have yet been calculated. See the *variable* doc page section on Variable Accuracy for more information on this topic.

1.118.5 Related commands

dump, *dump image*, *dump_modify*

1.118.6 Default

The defaults are listed on the doc pages for the *dump* and *dump image* and *dump_modify* commands.

1.119 write_restart command

1.119.1 Syntax

```
write_restart file keyword value ...
```

- file = name of file to write restart information to
- zero or more keyword/value pairs may be appended
- keyword = *fileper* or *nfile*

fileper arg = Np
  Np = write one file for every this many processors
nfile arg = Nf
  Nf = write this many files, one from each of Nf processors

1.119.2 Examples

```
write_restart restart.equil
write_restart poly.%.* nfile 10
```

1.119.3 Description

Write a binary restart file of the current state of the simulation.

During a long simulation, the *restart* command is typically used to output restart files periodically. The write_restart command is useful after a minimization or whenever you wish to write out a single current restart file.

Similar to *dump* files, the restart filename can contain two wild-card characters. If a "*" appears in the filename, it is replaced with the current timestep value. If a "%" character appears in the filename, then one file is written by each processor and the "%" character is replaced with the processor ID from 0 to P-1. An additional file with the "%" replaced by "base" is also written, which contains global information. For example, the files written for filename restart.% would be restart.base, restart.0, restart.1, ... restart.P-1. This creates smaller files and can be a fast mode of output and subsequent input on parallel machines that support parallel I/O. The optional *fileper* and *nfile* keywords discussed below can alter the number of files written.

Restart files can be read by a *read_restart* command to restart a simulation from a particular state. Because the file is binary (to enable exact restarts), it may not be readable on another machine. In this case, you can use the *-r command-line switch* to convert a restart file to a data file.

Note

Although the purpose of restart files is to enable restarting a simulation from where it left off, not all information about a simulation is stored in the file. For example, the list of fixes that were specified during the initial run is not stored, which means the new input script must specify any fixes you want to use. Even when restart information is stored in the file, as it is for some fixes, commands may need to be re-specified in the new input script, in order to re-use that information. Details are usually given in the documentation of the respective command. Also, see the *read_restart* command for general information about what is stored in a restart file.

The optional *nfile* or *fileper* keywords can be used in conjunction with the "%" wildcard character in the specified restart file name. As explained above, the "%" character causes the restart file to be written in pieces, one piece for each of P processors. By default P = the number of processors the simulation is running on. The *nfile* or *fileper* keyword can be used to set P to a smaller value, which can be more efficient when running on a large number of processors.

The *nfile* keyword sets P to the specified Nf value. For example, if Nf = 4, and the simulation is running on 100 processors, 4 files will be written, by processors 0,25,50,75. Each will collect information from itself and the next 24 processors and write it to a restart file.

For the *fileper* keyword, the specified value of Np means write one file for every Np processors. For example, if Np = 4, every fourth processor (0,4,8,12,etc) will collect information from itself and the next 3 processors and write it to a restart file.

1.119.4 Restrictions

This command requires inter-processor communication to migrate atoms before the restart file is written. This means that your system must be ready to perform a simulation before using this command (force fields setup, atom masses initialized, etc).

1.119.5 Related commands

restart, *read_restart*, *write_data*

1.119.6 Default

none