**Part II**

Programmer Guide

## 3.3.11 Testing (optional)

If your contribution contains new utility functions or a supporting class (i.e. anything that does not depend on a LAMMPS object), new unit tests should be added to a suitable folder in the unittest tree. When adding a new LAMMPS style computing forces or selected fixes, a .yaml file with a test configuration and reference data should be added for the styles where a suitable tester program already exists (e.g. pair styles, bond styles, etc.). Please see *this section in*

*the manual* for more information on how to enable, run, and expand testing.

# LAMMPS programming style

The aim of the LAMMPS developers is to use a consistent programming style and naming conventions across the entire code base, as this helps with maintenance, debugging, and understanding the code, both for developers and users. This page provides a list of standard style choices used in LAMMPS. Some of these standards are required, while others are just preferred. Following these conventions will make it much easier to integrate your contribution. If you are uncertain, please ask.

The files *pair\_lj\_cut.h*, *pair\_lj\_cut.cpp*, *utils.h*, and *utils.cpp* may serve as representative examples.

## Include files (varied)

* + - * Header files that define a new LAMMPS style (i.e. that have a SomeStyle(some/name,SomeName); macro in them) should only use the include file for the base class and otherwise use forward declarations and pointers; when interfacing to a library use the PIMPL (pointer to implementation) approach where you have a pointer to

a struct that contains all library specific data (and thus requires the library header) but use a forward declaration and define the struct only in the implementation file. This is a **strict** requirement since this is where type clashes between packages and hard-to-find bugs have regularly manifested in the past.

* + - * Header files, especially those defining a “style”, should only use the absolute minimum number of include files and **must not** contain any using statements. Typically, that would only be the header for the base class. Instead, any include statements should be put in the corresponding implementation files and forward declarations be used. For implementation files, the “include what you use” principle should be employed. However, there is the notable exception that when the pointers.h header is included (or the header of one of the classes derived from

it), certain headers will *always* be included and thus do not need to be explicitly specified. These are: *mpi.h*,

*cstddef*, *cstdio*, *cstdlib*, *string*, *utils.h*, *vector*, *fmt/format.h*, *climits*, *cinttypes*. This also means any such file can assume that *FILE*, *NULL*, and *INT\_MAX* are defined.

* + - * Class members variables should not be initialized in the header file, but instead should be initialized either in the initializer list of the constructor or explicitly assigned in the body of the constructor. If the member variable is relevant to the functionality of a class (for example when it stores a value from a command line argument), the

member variable declaration is followed by a brief comment explaining its purpose and what its values can be. Class members that are pointers should always be initialized to nullptr in the initializer list of the constructor. This reduces clutter in the header and avoids accessing uninitialized pointers, which leads to hard to debug issues, class members are often implicitly initialized to NULL on the first use (but *not* after a *clear command*). Please see the files reset\_atoms\_mol.h and reset\_atoms\_mol.cpp as an example.

* + - * System headers or headers from installed libraries are included with angular brackets (example: #include <vec- tor>), while local include files use double quotes (example: #include "atom.h")
      * When including system header files from the C library use the C++-style names (<cstdlib> or <cstring>) instead of the C-style names (<stdlib.h> or <string.h>)
      * The order of #include statements in a file some\_name.cpp that implements a class SomeName defined in a header file some\_name.h should be as follows:
        + #include "some\_name.h" followed by an empty line
        + LAMMPS include files e.g. #include "comm.h" or #include "modify.h" in alphabetical order followed by an empty line
        + System header files from the C++ or C standard library followed by an empty line
        + using namespace LAMMPS\_NS or other namespace imports.

## Whitespace (preferred)

Source files should not contain TAB characters unless required by the syntax (e.g. in makefiles) and no trailing whites- pace. Text files should have Unix-style line endings (LF-only). Git will automatically convert those in both directions when running on Windows; use dos2unix on Linux machines to convert files to Unix-style line endings. The last line of text files include a line ending.

You can check for these issues with the python scripts in the *“tools/coding\_standard”* folder. When run normally with a source file or a source folder as argument, they will list all non-conforming lines. By adding the *-f* flag to the command line, they will modify the flagged files to try to remove the detected issues.

## Constants (strongly preferred)

Global or per-file constants should be declared as *static constexpr* variables rather than via the pre-processor with

*#define*. The name of constants should be all uppercase. This has multiple advantages:

* + - * constants are easily identified as such by their all upper case name
      * rather than a pure text substitution during pre-processing, *constexpr variables* have a type associated with them and are processed later in the parsing process where the syntax checks and type specific processing (e.g. via overloads) can be applied to them.
      * compilers can emit a warning if the constant is not used and thus can be removed (we regularly check for and remove dead code like this)
      * there are no unexpected substitutions and thus confusing syntax errors when compiling leading to, for instance, conflicts so that LAMMPS cannot be compiled with certain combinations of packages (this *has* happened mul- tiple times in the past).

Pre-processor defines should be limited to macros (but consider C++ templates) and conditional compilation. If a per- processor define must be used, it should be defined at the top of the .cpp file after the include statements and at all cost it should be avoided to put them into header files.

Some sets of commonly used constants are provided in the MathConst and EwaldConst namespaces and implemented in the files math\_const.h and ewald\_const.h, respectively.

There are always exceptions, special cases, and legacy code in LAMMPS, so please contact the LAMMPS developers if you are not sure.

## Placement of braces (strongly preferred)

For new files added to the “src” tree, a [clang-format](https://clang.llvm.org/docs/ClangFormat.html) configuration file is provided under the name *.clang-format*. This file is compatible with clang-format version 8 and later. With that file present, files can be reformatted according to the configuration with a command like: *clang-format -i new-file.cpp*. Ideally, this is done while writing the code or before a pull request is submitted. Blocks of code where the reformatting from clang-format yields hard-to-read or otherwise undesirable output may be protected with placing a pair *// clang-format off* and *// clang-format on* comments around that block.

## Miscellaneous standards (varied)

* + - * I/O is done via the C-style stdio library and **not** iostreams.
      * Do not use so-called “alternative tokens” like and, or, not and similar, but rather use the corresponding operators

&&, ||, and !. The alternative tokens are not available by default on all compilers.

* + - * Output to the screen and the logfile should use the corresponding FILE pointers and only be done on MPI rank

0. Use the utils::logmesg() convenience function where possible.

* + - * Usage of C++11 *virtual*, *override*, *final* keywords: Please follow the [C++ Core Guideline C.128](https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-override). That means, you should only use *virtual* to declare a new virtual function, *override* to indicate you are overriding an existing virtual function, and *final* to prevent any further overriding.
      * Trivial destructors: Do not write destructors when they are empty and *default*.

// don't write destructors for A or B like this

class A : protected Pointers { public:

A();

~A() override {}

};

class B : protected Pointers { public:

B();

~B() override = default;

};

// instead, let the compiler create the implicit default destructor by not writing it class A : protected Pointers {

public:

A();

};

class B : protected Pointers { public:

B();

};

* + - * Please use clang-format only to reformat files that you have contributed. For header files containing a SomeStyle(keyword, ClassName) macros it is required to have this macro embedded with a pair of // clang-format off, // clang-format on comments and the line must be terminated with a semicolon (;). Ex- ample:

#ifdef COMMAND\_CLASS

// clang-format off CommandStyle(run,Run);

// clang-format on #else

#ifndef LMP\_RUN\_H [...]

You may also use // clang-format on/off throughout your files to protect individual sections from being refor- matted.

* + - * All files should have 0644 permissions, i.e. writable by the user only and readable by all and no executable permissions. Executable permissions (0755) should only be for shell scripts or python or similar scripts for interpreted script languages.

# Atom styles

Classes that define an *atom style* are derived from the AtomVec class and managed by the Atom class. The atom style determines what attributes are associated with an atom and communicated when it is a ghost atom or migrates to a new processor. A new atom style can be created if one of the existing atom styles does not define all the attributes you need to store and communicate with atoms.

The file atom\_vec\_atomic.cpp is the simplest example of an atom style. Examining the code for others will make these instructions more clear.

Note that the *atom style hybrid* command can be used to define atoms or particles which have the union of properties of individual styles. Also the *fix property/atom* command can be used to add a single property (e.g. charge or a molecule ID) to a style that does not have it. It can also be used to add custom properties to an atom, with options to communicate them with ghost atoms or read them from a data file. Other LAMMPS commands can access these custom properties, as can new pair, fix, compute styles that are written to work with these properties. For example, the *set* command can be used to set the values of custom per-atom properties from an input script. All of these methods are less work than writing and testing(!) code for a new atom style.

If you follow these directions your new style will automatically work in tandem with others via the *atom\_style hybrid*

command.

The first step is to define a set of string lists in the constructor of the new derived class. Each list will have zero or more comma-separated strings that correspond to the variable names used in the atom.h header file for per-atom properties. Note that some represent per-atom vectors (q, molecule) while other are per-atom arrays (x,v). For all but the last two lists you do not need to specify any of (id,type,x,v,f). Those are included automatically as needed in the other lists.

|  |  |
| --- | --- |
| fields\_grow | full list of properties which is allocated and stored |
| fields\_copy | list of properties to copy atoms are rearranged on-processor |
| fields\_comm | list of properties communicated to ghost atoms every step |
| fields\_comm\_vel | additional properties communicated if *comm\_modify vel* is used |
| fields\_reverse | list of properties summed from ghost atoms every step |
| fields\_border | list of properties communicated with ghost atoms every reneighboring step |
| fields\_border\_vel | additional properties communicated if *comm\_modify vel* is used |
| fields\_exchange | list of properties communicated when an atom migrates to another processor |
| fields\_restart | list of properties written/read to/from a restart file |
| fields\_create | list of properties defined when an atom is created by *create\_atoms* |
| fields\_data\_atom | list of properties (in order) in the Atoms section of a data file, as read by *read\_data* |
| fields\_data\_vel | list of properties (in order) in the Velocities section of a data file, as read by *read\_data* |

In these lists you can list variable names which LAMMPS already defines (in some other atom style), or you can create new variable names. You should not re-use a LAMMPS variable in your atom style that is used for something with a different meaning in another atom style. If the meaning is related, but interpreted differently by your atom style, then using the same variable name means a user must not use your style and the other style together in a *atom\_style hybrid* command. Because there will only be one value of the variable and different parts of LAMMPS will then likely use it differently. LAMMPS has no way of checking for this.

If you are defining new variable names then make them descriptive and unique to your new atom style. For example choosing “e” for energy is a bad choice; it is too generic. A better choice would be “e\_foo”, where “foo” is specific to your style.

If any of the variable names in your new atom style do not exist in LAMMPS, you need to add them to the src/atom.h and atom.cpp files.

Search for the word “customize” or “customization” in these 2 files to see where to add your variable. Adding a flag to the 2nd customization section in atom.h is only necessary if your code (e.g. a pair style) needs to check that a per-atom property is defined. These flags should also be set in the constructor of the atom style child class.

In atom.cpp, aside from the constructor and destructor, there are 3 methods that a new variable name or flag needs to be added to.

In Atom::peratom\_create() when using the Atom::add\_peratom() method, a cols argument of 0 is for per-atom vectors, a length > 1 is for per-atom arrays. Note the use of the extra per-thread flag and the add\_peratom\_vary() method when the last dimension of the array is variable-length.

Adding the variable name to Atom::extract() enables the per-atom data to be accessed through the *LAMMPS library interface* by a calling code, including from *Python*.

The constructor of the new atom style will also typically set a few flags which are defined at the top of atom\_vec.h. If these are unclear, see how other atom styles use them.

The grow\_pointers() method is also required to make a copy of peratom data pointers, as explained in the code.

There are a number of other optional methods which your atom style can implement. These are only needed if you need to do something out-of-the-ordinary which the default operation of the AtomVec parent class does not take care of. The best way to figure out why they are sometimes useful is to look at how other atom styles use them.

* process\_args = use if the atom style has arguments
* init = called before each run
* force\_clear = called before force computations each timestep

A few atom styles define “bonus” data associated with some or all of their particles, such as *atom\_style ellipsoid or tri*. These methods work with that data:

* copy\_bonus
* clear\_bonus
* pack\_comm\_bonus
* unpack\_comm\_bonus
* pack\_border\_bonus
* unpack\_border\_bonus
* pack\_exchange\_bonus
* unpack\_exchange\_bonus
* size\_restart\_bonus
* pack\_restart\_bonus
* unpack\_restart\_bonus
* data\_atom\_bonus
* memory\_usage\_bonus

The *atom\_style body* command can define a particle geometry with an arbitrary number of values. This method reads it from a data file:

* data\_body

These methods are called before or after operations handled by the parent AtomVec class. They allow an atom style to do customized operations on the per-atom values. For example *atom\_style sphere* reads a diameter and density of each particle from a data file. But these need to be converted internally to a radius and mass. That operation is done in the data\_atom\_post() method.

* pack\_restart\_pre
* pack\_restart\_post
* unpack\_restart\_init
* create\_atom\_post
* data\_atom\_post
* pack\_data\_pre
* pack\_data\_post

These methods enable the *compute property/atom* command to access per-atom variables it does not already define as arguments, so that they can be written to a dump file or used by other LAMMPS commands.

* property\_atom
* pack\_property\_atom

# Pair styles

Classes that compute pairwise non-bonded interactions are derived from the Pair class. In LAMMPS, pairwise force calculations include many-body potentials such as EAM, Tersoff, or ReaxFF where particles interact without an explicit bond topology but include interactions beyond pairwise non-bonded contributions. New styles can be created to add

support for additional pair potentials to LAMMPS. When the modifications are small, sometimes it is more effective to derive from an existing pair style class. This latter approach is also used by *Accelerator packages* where the accelerated style names differ from their base classes by an appended suffix.

The file src/pair\_lj\_cut.cpp is an example of a Pair class with a very simple potential function. It includes several optional methods to enable its use with *run\_style respa* and *compute group/group*. *Writing new pair styles* contains a detailed discussion of writing new pair styles from scratch, and how simple and more complex pair styles can be

implemented with examples from existing pair styles.

Here is a brief list of some the class methods in the Pair class that *must* be or *may* be overridden in a derived class for a new pair style.

|  |  |
| --- | --- |
| Required | “pure” methods that *must* be overridden in a derived class |
| compute | workhorse routine that computes pairwise interactions |
| settings | processes the arguments to the pair\_style command |
| coeff | set coefficients for one i,j type pair, called from pair\_coeff |

|  |  |
| --- | --- |
| Optional | methods that have a default or dummy implementation |
| init\_one | perform initialization for one i,j type pair |
| init\_style | style initialization: request neighbor list(s), error checks |
| init\_list | Neighbor class callback function to pass neighbor list to pair style |
| single | force/r and energy of a single pairwise interaction between two atoms |
| compute\_inner/middle/outer | versions of compute used by rRESPA |
| memory\_usage | return estimated amount of memory used by the pair style |
| modify\_params | process arguments to pair\_modify command |
| extract | provide access to internal scalar or per-type data like cutoffs |
| extract\_peratom | provide access to internal per-atom data |
| setup | initialization at the beginning of a run |
| finish | called at the end of a run, e.g. to print |
| write & read\_restart | write/read i,j pair coeffs to restart files |
| write & read\_restart\_settings | write/read global settings to restart files |
| write\_data | write Pair Coeffs section to data file |
| write\_data\_all | write PairIJ Coeffs section to data file |
| pack & unpack\_forward\_comm | copy data to and from buffer if style uses forward communication |
| pack & unpack\_reverse\_comm | copy data to and from buffer if style uses reverse communication |
| reinit | reset all type-based parameters, called by fix adapt for example |
| reset\_dt | called when the time step is changed by timestep or fix reset/dt |

Here is a list of flags or settings that should be set in the constructor of the derived pair class when they differ from the default setting.

|  |  |  |
| --- | --- | --- |
| Name of flag | Description | default |
| single\_enable | 1 if single() method is implemented, 0 if missing | 1 |
| respa\_enable | 1 if pair style has compute\_inner/middle/outer() | 0 |
| restartinfo | 1 if pair style writes its settings to a restart | 1 |
| one\_coeff | 1 if only a pair\_coeff \* \* command is allowed | 0 |
| manybody\_flag | 1 if pair style is a manybody potential | 0 |
| unit\_convert\_flag | value != 0 indicates support for unit conversion | 0 |
| no\_virial\_fdotr\_compute | 1 if pair style does not call virial\_fdotr\_compute() | 0 |
| writedata | 1 if write\_data() and write\_data\_all() are implemented | 0 |
| comm\_forward | size of buffer (in doubles) for forward communication | 0 |
| comm\_reverse | size of buffer (in doubles) for reverse communication | 0 |
| ghostneigh | 1 if cutghost is set and style uses neighbors of ghosts | 0 |
| finitecutflag | 1 if cutoff depends on diameter of atoms | 0 |
| reinitflag | 1 if style has reinit() and is compatible with fix adapt | 0 |
| ewaldflag | 1 if compatible with kspace\_style ewald | 0 |
| pppmflag | 1 if compatible with kspace\_style pppm | 0 |
| msmflag | 1 if compatible with kspace\_style msm | 0 |
| dispersionflag | 1 if compatible with ewald/disp or pppm/disp | 0 |
| tip4pflag | 1 if compatible with kspace\_style pppm/tip4p | 0 |
| dipoleflag | 1 if compatible with dipole kspace\_style | 0 |
| spinflag | 1 if compatible with spin kspace\_style | 0 |

# Bond, angle, dihedral, improper styles

Classes that compute molecular interactions are derived from the Bond, Angle, Dihedral, and Improper classes. New styles can be created to add new potentials to LAMMPS.

Bond\_harmonic.cpp is the simplest example of a bond style. Ditto for the harmonic forms of the angle, dihedral, and improper style commands.

Here is a brief description of common methods you define in your new derived class. See bond.h, angle.h, dihedral.h, and improper.h for details and specific additional methods.

|  |  |
| --- | --- |
| Required | “pure” methods that *must* be overridden in a derived class |
| compute | compute the molecular interactions for all listed items |
| coeff | set coefficients for one type |
| equilibrium\_distance | length of bond, used by SHAKE (bond styles only) |
| equilibrium\_angle | opening of angle, used by SHAKE (angle styles only) |
| write & read\_restart | writes/reads coeffs to restart files |
| single | force/r (bond styles only) and energy of a single bond or angle |

|  |  |
| --- | --- |
| Optional | methods that have a default or dummy implementation |
| init | check if all coefficients are set, calls init\_style() |
| init\_style | check if style specific conditions are met |
| settings | apply global settings for all types |
| write & read\_restart\_settings | writes/reads global style settings to restart files |
| write\_data | write corresponding Coeffs section(s) in data file |
| memory\_usage | tally memory allocated by the style |
| extract | provide access to internal data (bond or angle styles only) |
| reinit | reset all type-based parameters, called by fix adapt (bonds only) |
| pack & unpack\_forward\_comm | copy data to and from buffer in forward communication (bonds only) |
| pack & unpack\_reverse\_comm | copy data to and from buffer in reverse communication (bonds only) |

Here is a list of flags or settings that should be set in the constructor of the derived class when they differ from the default setting.

|  |  |  |
| --- | --- | --- |
| Name of flag | Description | default |
| writedata | 1 if write\_data() is implemented | 1 |
| single\_extra | number of extra single values calculated (bond styles only) | 0 |
| partial\_flag | 1 if bond type can be set to 0 and deleted (bond styles only) | 0 |
| reinitflag | 1 if style has reinit() and is compatible with fix adapt | 1 |
| comm\_forward | size of buffer (in doubles) for forward communication (bond styles only) | 0 |
| comm\_reverse | size of buffer (in doubles) for reverse communication (bond styles only) | 0 |
| comm\_reverse\_off | size of buffer for reverse communication with newton off (bond styles only) | 0 |

# Compute styles

Classes that compute scalar and vector quantities like temperature and the pressure tensor, as well as classes that compute per-atom quantities like kinetic energy and the centro-symmetry parameter are derived from the Compute class. New styles can be created to add new calculations to LAMMPS.

Compute\_temp.cpp is a simple example of computing a scalar temperature. Compute\_ke\_atom.cpp is a simple example of computing per-atom kinetic energy.

Here is a brief description of methods you define in your new derived class. See compute.h for details.

|  |  |
| --- | --- |
| init | perform one time setup (required) |
| init\_list | neighbor list setup, if needed (optional) |
| compute\_scalar | compute a scalar quantity (optional) |
| compute\_vector | compute a vector of quantities (optional) |
| compute\_peratom | compute one or more quantities per atom (optional) |
| compute\_local | compute one or more quantities per processor (optional) |
| pack\_comm | pack a buffer with items to communicate (optional) |
| unpack\_comm | unpack the buffer (optional) |
| pack\_reverse | pack a buffer with items to reverse communicate (optional) |
| unpack\_reverse | unpack the buffer (optional) |
| remove\_bias | remove velocity bias from one atom (optional) |
| remove\_bias\_all | remove velocity bias from all atoms in group (optional) |
| restore\_bias | restore velocity bias for one atom after remove\_bias (optional) |
| restore\_bias\_all | same as before, but for all atoms in group (optional) |
| pair\_tally\_callback | callback function for *tally*-style computes (optional). |
| memory\_usage | tally memory usage (optional) |

Tally-style computes are a special case, as their computation is done in two stages: the callback function is registered with the pair style and then called from the Pair::ev\_tally() function, which is called for each pair after force and energy has been computed for this pair. Then the tallied values are retrieved with the standard compute\_scalar or compute\_vector or compute\_peratom methods. The *compute styles in the TALLY package* provide *examples* for utilizing this mechanism.

# Fix styles

In LAMMPS, a “fix” is any operation that is computed during timestepping that alters some property of the system. Essentially everything that happens during a simulation besides force computation, neighbor list construction, and output, is a “fix”. This includes time integration (update of coordinates and velocities), force constraints or boundary conditions (SHAKE or walls), and diagnostics (compute a diffusion coefficient). New styles can be created to add new options to LAMMPS.

Fix\_setforce.cpp is a simple example of setting forces on atoms to prescribed values. There are dozens of fix options already in LAMMPS; choose one as a template that is similar to what you want to implement.

Here is a brief description of methods you can define in your new derived class. See fix.h for details.

|  |  |
| --- | --- |
| setmask | determines when the fix is called during the timestep (required) |
| init | initialization before a run (optional) |
| init\_list | store pointer to neighbor list; called by neighbor list code (optional) |
| setup\_pre\_exchange | called before atom exchange in setup (optional) |

continues on next page

Table 1 – continued from previous page

|  |  |
| --- | --- |
| setup\_pre\_force | called before force computation in setup (optional) |
| setup | called immediately before the first timestep and after forces are computed (optional) |
| min\_setup\_pre\_force | like setup\_pre\_force, but for minimizations instead of MD runs (optional) |
| min\_setup | like setup, but for minimizations instead of MD runs (optional) |
| initial\_integrate | called at very beginning of each timestep (optional) |
| pre\_exchange | called before atom exchange on re-neighboring steps (optional) |
| pre\_neighbor | called before neighbor list build (optional) |
| pre\_force | called before pair & molecular forces are computed (optional) |
| post\_force | called after pair & molecular forces are computed and communicated (optional) |
| final\_integrate | called at end of each timestep (optional) |
| end\_of\_step | called at very end of timestep (optional) |
| write\_restart | dumps fix info to restart file (optional) |
| restart | uses info from restart file to re-initialize the fix (optional) |
| grow\_arrays | allocate memory for atom-based arrays used by fix (optional) |
| copy\_arrays | copy atom info when an atom migrates to a new processor (optional) |
| pack\_exchange | store atom’s data in a buffer (optional) |
| unpack\_exchange | retrieve atom’s data from a buffer (optional) |
| pack\_restart | store atom’s data for writing to restart file (optional) |
| unpack\_restart | retrieve atom’s data from a restart file buffer (optional) |
| size\_restart | size of atom’s data (optional) |
| maxsize\_restart | max size of atom’s data (optional) |
| setup\_pre\_force\_respa | same as setup\_pre\_force, but for rRESPA (optional) |
| initial\_integrate\_respa | same as initial\_integrate, but for rRESPA (optional) |
| post\_integrate\_respa | called after the first half integration step is done in rRESPA (optional) |
| pre\_force\_respa | same as pre\_force, but for rRESPA (optional) |
| post\_force\_respa | same as post\_force, but for rRESPA (optional) |
| final\_integrate\_respa | same as final\_integrate, but for rRESPA (optional) |
| min\_pre\_force | called after pair & molecular forces are computed in minimizer (optional) |
| min\_post\_force | called after pair & molecular forces are computed and communicated in minimizer (optional) |
| min\_store | store extra data for linesearch based minimization on a LIFO stack (optional) |
| min\_pushstore | push the minimization LIFO stack one element down (optional) |
| min\_popstore | pop the minimization LIFO stack one element up (optional) |
| min\_clearstore | clear minimization LIFO stack (optional) |
| min\_step | reset or move forward on line search minimization (optional) |
| min\_dof | report number of degrees of freedom *added* by this fix in minimization (optional) |
| max\_alpha | report maximum allowed step size during linesearch minimization (optional) |
| pack\_comm | pack a buffer to communicate a per-atom quantity (optional) |
| unpack\_comm | unpack a buffer to communicate a per-atom quantity (optional) |
| pack\_reverse\_comm | pack a buffer to reverse communicate a per-atom quantity (optional) |
| unpack\_reverse\_comm | unpack a buffer to reverse communicate a per-atom quantity (optional) |
| dof | report number of degrees of freedom *removed* by this fix during MD (optional) |
| compute\_scalar | return a global scalar property that the fix computes (optional) |
| compute\_vector | return a component of a vector property that the fix computes (optional) |
| compute\_array | return a component of an array property that the fix computes (optional) |
| deform | called when the box size is changed (optional) |
| reset\_target | called when a change of the target temperature is requested during a run (optional) |
| reset\_dt | is called when a change of the time step is requested during a run (optional) |
| modify\_param | called when a fix\_modify request is executed (optional) |
| memory\_usage | report memory used by fix (optional) |
| thermo | compute quantities for thermodynamic output (optional) |

Typically, only a small fraction of these methods are defined for a particular fix. Setmask is mandatory, as it deter-

mines when the fix will be invoked during the timestep. Fixes that perform time integration (*nve*, *nvt*, *npt*) imple- ment initial\_integrate() and final\_integrate() to perform velocity Verlet updates. Fixes that constrain forces implement post\_force().

Fixes that perform diagnostics typically implement end\_of\_step(). For an end\_of\_step fix, one of your fix arguments must be the variable “nevery” which is used to determine when to call the fix and you must set this variable in the constructor of your fix. By convention, this is the first argument the fix defines (after the ID, group-ID, style).

If the fix needs to store information for each atom that persists from timestep to timestep, it can manage that memory and migrate the info with the atoms as they move from processors to processor by implementing the grow\_arrays, copy\_arrays, pack\_exchange, and unpack\_exchange methods. Similarly, the pack\_restart and unpack\_restart methods can be implemented to store information about the fix in restart files. If you wish an integrator or force constraint fix to work with rRESPA (see the *run\_style* command), the initial\_integrate, post\_force\_integrate, and final\_integrate\_respa methods can be implemented. The thermo method enables a fix to contribute values to thermodynamic output, as printed quantities and/or to be summed to the potential energy of the system.

# Input script command style

New commands can be added to LAMMPS input scripts by adding new classes that are derived from the Command class and thus must have a “command” method. For example, the create\_atoms, read\_data, velocity, and run commands are all implemented in this fashion. When such a command is encountered in the LAMMPS input script, LAMMPS simply creates a class instance with the corresponding name, invokes the “command” method of the class, and passes it the arguments from the input script. The command method can perform whatever operations it wishes on LAMMPS data structures.

The single method your new class must define is as follows:

command operations performed by the new command

Of course, the new class can define other methods and variables as needed.

# Dump styles

Classes that dump per-atom info to files are derived from the Dump class. To dump new quantities or in a new format, a new derived dump class can be added, but it is typically simpler to modify the DumpCustom class contained in the dump\_custom.cpp file.

Dump\_atom.cpp is a simple example of a derived dump class.

Here is a brief description of methods you define in your new derived class. See dump.h for details.

|  |  |
| --- | --- |
| write\_header | write the header section of a snapshot of atoms |
| count | count the number of lines a processor will output |
| pack | pack a proc’s output data into a buffer |
| write\_data | write a proc’s data to a file |

See the *dump* command and its *custom* style for a list of keywords for atom information that can already be dumped by DumpCustom. It includes options to dump per-atom info from Compute classes, so adding a new derived Compute class is one way to calculate new quantities to dump.

Note that new keywords for atom properties are not typically added to the *dump custom* command. Instead they are added to the *compute property/atom* command.

**3.10. Input script command style** **759**

# Kspace styles

Classes that compute long-range Coulombic interactions via K-space representations (Ewald, PPPM) are derived from the KSpace class. New styles can be created to add new K-space options to LAMMPS.

Ewald.cpp is an example of computing K-space interactions.

Here is a brief description of methods you define in your new derived class. See kspace.h for details.

|  |  |
| --- | --- |
| init | initialize the calculation before a run |
| setup | computation before the first timestep of a run |
| compute | every-timestep computation |
| memory\_usage | tally of memory usage |

# Minimization styles

Classes that perform energy minimization derived from the Min class. New styles can be created to add new minimiza- tion algorithms to LAMMPS.

Min\_cg.cpp is an example of conjugate gradient minimization.

Here is a brief description of methods you define in your new derived class. See min.h for details.

|  |
| --- |
| init initialize the minimization before a run |
| run perform the minimization |
| memory\_usage tally of memory usage |

# Region styles

Classes that define geometric regions are derived from the Region class. Regions are used elsewhere in LAMMPS to group atoms, delete atoms to create a void, insert atoms in a specified region, etc. New styles can be created to add new region shapes to LAMMPS.

Region\_sphere.cpp is an example of a spherical region.

Here is a brief description of methods you define in your new derived class. See region.h for details.

|  |  |
| --- | --- |
| inside | determine whether a point is in the region |
| surface\_interior | determine if a point is within a cutoff distance inside of surface |
| surface\_exterior | determine if a point is within a cutoff distance outside of surface |
| shape\_update | change region shape if set by time-dependent variable |

# Body styles

Classes that define body particles are derived from the Body class. Body particles can represent complex entities, such as surface meshes of discrete points, collections of sub-particles, deformable objects, etc.

See the *Howto body* page for an overview of using body particles and the various body styles LAMMPS supports. New styles can be created to add new kinds of body particles to LAMMPS.

Body\_nparticle.cpp is an example of a body particle that is treated as a rigid body containing N sub-particles. Here is a brief description of methods you define in your new derived class. See body.h for details.

|  |  |
| --- | --- |
| data\_body | process a line from the Bodies section of a data file |
| noutrow | number of sub-particles output is generated for |
| noutcol | number of values per-sub-particle output is generated for |
| output | output values for the Mth sub-particle |
| pack\_comm\_body | body attributes to communicate every timestep |
| unpack\_comm\_body | unpacking of those attributes |
| pack\_border\_body | body attributes to communicate when reneighboring is done |
| unpack\_border\_body | unpacking of those attributes |

# Granular Sub-Model styles

In granular models, particles are spheres with a finite radius and rotational degrees of freedom as further described in the *Howto granular page*. Interactions between pair of particles or particles and walls may therefore depend on many different modes of motion as described in *pair granular* and *fix wall/gran*. In both cases, the exchange of forces, torques, and heat flow between two types of bodies is defined using a GranularModel class. The GranularModel class organizes the details of an interaction using a series of granular sub-models each of which describe a particular interaction mode (e.g. normal forces or rolling friction). From a parent GranSubMod class, several types of sub-model classes are derived:

* GranSubModNormal: normal force sub-model
* GranSubModDamping: normal damping sub-model
* GranSubModTangential: tangential forces and sliding friction sub-model
* GranSubModRolling: rolling friction sub-model
* GranSubModTwisting: twisting friction sub-model
* GranSubModHeat: heat conduction sub-model

For each type of sub-model, more classes are further derived, each describing a specific implementation. For instance, from the GranSubModNormal class the GranSubModNormalHooke, GranSubModNormalHertz, and GranSubMod- NormalJKR classes are derived which calculate Hookean, Hertzian, or JKR normal forces, respectively. This modular structure simplifies the addition of new granular contact models as one only needs to create a new GranSubMod class without having to modify the more complex PairGranular, FixGranWall, and GranularModel classes. Most GranSub- Mod methods are also already defined by the parent classes so new contact models typically only require edits to a few relevant methods (e.g. methods that define coefficients and calculate forces).

Each GranSubMod class has a pointer to both the LAMMPS class and the GranularModel class which owns it, lmp and gm, respectively. The GranularModel class includes several public variables that describe the geometry/dynamics of the contact such as

|  |  |
| --- | --- |
| xi and xj | Positions of the two contacting bodies |
| vi and vj | Velocities of the two contacting bodies |
| omegai and omegaj | Angular velocities of the two contacting bodies |
| dx and nx | The displacement and normalized displacement vectors |
| r, rsq, and rinv | The distance, distance squared, and inverse distance |
| radsum | The sum of particle radii |
| vr, vn, and vt | The relative velocity vector and its normal and tangential components |
| wr | The relative rotational velocity |

These quantities, among others, are calculated in the GranularModel->check\_contact() and

GranularModel->calculate\_forces() methods which can be referred to for more details.

To create a new GranSubMod class, it is recommended that one first looks at similar GranSubMod classes. All GranSub- Mod classes share several general methods which may need to be defined

|  |
| --- |
| mix\_coeff() Optional method to define how coefficients are mixed for different atom types. By default, coefficients are mixed using a geometric mean. |
| coeffs\_to\_local() Parses coefficients to define local variables. Run once at model construction. |
| init() Optional method to define local variables after other GranSubMod types were created.  For instance, this method may be used by a tangential model that derives parameters  from the normal model. |

The Normal, Damping, Tangential, Twisting, and Rolling sub-models also have a calculate\_forces() method which calculate the respective forces/torques. Correspondingly, the Heat sub-model has a calculate\_heat() method. Lastly, the Normal sub-model has a few extra optional methods:

|  |  |
| --- | --- |
| touch() | Tests whether particles are in contact. By default, when particles overlap. |
| pulloff\_distance() | Returns the distance at which particles stop interacting. By default, when particles no  longer overlap. |
| calculate\_radius() | Returns the radius of the contact. By default, the radius of the geometric cross section. |
| set\_fncrit() | Defines the critical force to break the contact used by some tangential, rolling, and  twisting sub-models. By default, the current total normal force including damping. |

As an example, say one wanted to create a new normal force option that consisted of a Hookean force with a piecewise stiffness. This could be done by adding a new set of files gran\_sub\_mod\_custom.h:

#ifdef GranSubMod\_CLASS

// clang-format off GranSubModStyle(hooke/piecewise,GranSubModNormalHookePiecewise,NORMAL);

// clang-format on #else

#ifndef GRAN\_SUB\_MOD\_CUSTOM\_H\_ #define GRAN\_SUB\_MOD\_CUSTOM\_H\_

#include "gran\_sub\_mod.h" #include "gran\_sub\_mod\_normal.h"

namespace LAMMPS\_NS { namespace Granular\_NS {

(continues on next page)

(continued from previous page)

class GranSubModNormalHookePiecewise : public GranSubModNormal { public:

GranSubModNormalHookePiecewise(class GranularModel \*, class LAMMPS \*); void coeffs\_to\_local() override;

double calculate\_forces() override; protected:

double k1, k2, delta\_switch;

};

} // namespace Granular\_NS

} // namespace LAMMPS\_NS

#endif /\*GRAN\_SUB\_MOD\_CUSTOM\_H\_ \*/ #endif /\*GRAN\_SUB\_MOD\_CLASS\_H\_ \*/

and gran\_sub\_mod\_custom.cpp

#include "gran\_sub\_mod\_custom.h" #include "gran\_sub\_mod\_normal.h" #include "granular\_model.h"

using namespace LAMMPS\_NS; using namespace Granular\_NS;

GranSubModNormalHookePiecewise::GranSubModNormalHookePiecewise(GranularModel \*gm,␣

*‹→*LAMMPS \*lmp) : GranSubModNormal(gm, lmp)

{

num\_coeffs = 4;

}

/\*

\*/

void GranSubModNormalHookePiecewise::coeffs\_to\_local()

{

k1 = coeffs[0]; k2 = coeffs[1];

damp = coeffs[2]; delta\_switch = coeffs[3];

}

/\*

\*/

double GranSubModNormalHookePiecewise::calculate\_forces()

{

double Fne;

if (gm->delta >= delta\_switch) {

Fne = k1 \* delta\_switch + k2 \* (gm->delta - delta\_switch);

} else {

Fne = k1 \* gm->delta;

}

return Fne;

}

# Thermodynamic output options

The Thermo class computes and prints thermodynamic information to the screen and log file; see the files thermo.cpp

and thermo.h.

There are four styles defined in thermo.cpp: “one”, “multi”, “yaml”, and “custom”. The “custom” style allows the user to explicitly list keywords for individual quantities to print when thermodynamic output is generated. The others

have a fixed list of keywords. See the *thermo\_style* command for a list of available quantities. The formatting of the “custom” style defaults to the “one” style, but can be adapted using *thermo\_modify line*.

The thermo styles (one, multi, etc) are defined by lists of keywords with associated formats for integer and floating point numbers and identified by an enumerator constant. Adding a new style thus mostly requires defining a new list of keywords and the associated formats and then inserting the required output processing where the enumerators are identified. Search for the word “CUSTOMIZATION” with references to “thermo style” in the thermo.cpp file to

see the locations where code will need to be added. The member function Thermo::header() prints output at the very

beginning of a thermodynamic output block and can be used to print column headers or other front matter. The member function Thermo::footer() prints output at the end of a thermodynamic output block. The formatting of the output is done by assembling a “line” (which may span multiple lines if the style inserts newline characters (”n” as in the “multi” style).

New thermodynamic keywords can also be added to thermo.cpp to compute new quantities for output. Search for the word “CUSTOMIZATION” with references to “keyword” in thermo.cpp to see the several locations where code will need to be added. Effectively, you need to define a member function that computes the property, add an if statement in Thermo::parse\_fields() where the corresponding header string for the keyword and the function pointer is registered by calling the Thermo::addfield() method, and add an if statement in Thermo::evaluate\_keyword() which is called from the Variable class when a thermo keyword is encountered.



**Note**

The third argument to Thermo::addfield() is a flag indicating whether the function for the keyword computes a floating point (FLOAT), regular integer (INT), or big integer (BIGINT) value. This information is used for formatting the thermodynamic output. Inside the function the result must then be stored either in the dvalue, ivalue or bivalue member variable, respectively.

Since the *thermo\_style custom* command allows to use output of quantities calculated by *fixes*, *computes*, and *variables*, it may often be simpler to compute what you wish via one of those constructs, rather than by adding a new keyword to the thermo\_style command.

# Variable options

The Variable class computes and stores *variable* information in LAMMPS; see the file variable.cpp. The value associ- ated with a variable can be periodically printed to the screen via the *print*, *fix print*, or *thermo\_style custom* commands. Variables of style “equal” can compute complex equations that involve the following types of arguments:

thermo keywords = ke, vol, atoms, ... other variables = v\_a, v\_myvar, ...

math functions = div(x,y), mult(x,y), add(x,y), ... group functions = mass(group), xcm(group,x), ... atom values = x[123], y[3], vx[34], ...

compute values = c\_mytemp[0], c\_thermo\_press[3], ...

Adding keywords for the *thermo\_style custom* command (which can then be accessed by variables) is discussed in the

[*Modify thermo*](#_bookmark1)documentation.

Adding a new math function of one or two arguments can be done by editing one section of the Variable::evaluate()

method. Search for the word “customize” to find the appropriate location.

Adding a new group function can be done by editing one section of the Variable::evaluate() method. Search for the word “customize” to find the appropriate location. You may need to add a new method to the Group class as well (see the group.cpp file).

Accessing a new atom-based vector can be done by editing one section of the Variable::evaluate() method. Search for the word “customize” to find the appropriate location.

Adding new *compute styles* (whose calculated values can then be accessed by variables) is discussed in the [*Modify*](#_bookmark0)[*compute*](#_bookmark0)documentation.