DSMC Users Manual

name of code

http://www.sandia.gov/~sjplimp/dsmc.html − Sandia National Laboratories

Table of Contents

Table of Contents

Table of Contents

DSMC Documentation

Version info:

The DSMC "version" is the date when it was released, such as 1 May 2010. DSMC is updated continuously. Whenever we fix a bug or add a feature, we release it immediately, and post a notice on this page of the WWW site. Each dated copy of DSMC contains all the features and bug−fixes up to and including that version date. The version date is printed to the screen and logfile every time you run DSMC. It is also in the file src/version.h and in the DSMC directory name created when you unpack a tarball.

- If you browse the HTML doc pages on the DSMC WWW site, they always describe the most current version of DSMC.
- If you browse the HTML doc pages included in your tarball, they describe the version you have.
- The PDF file on the WWW site or in the tarball is updated about once per month. This is because it is large, and we don't want it to be part of very patch.
- There is also a Developer.pdf file in the doc directory, which describes the internal structure and algorithms of DSMC.

DSMC stands for Large−scale Atomic/Molecular Massively Parallel Simulator.

DSMC is a classical molecular dynamics simulation code designed to run efficiently on parallel computers. It was developed at Sandia National Laboratories, a US Department of Energy facility, with funding from the DOE. It is an open−source code, distributed freely under the terms of the GNU Public License (GPL).

The primary developers of DSMC are Steve Plimpton, Aidan Thompson, and Paul Crozier who can be contacted at sjplimp,athomps,pscrozi at sandia.gov. The DSMC WWW Site at http://lammps.sandia.gov has more information about the code and its uses.

The DSMC documentation is organized into the following sections. If you find errors or omissions in this manual or have suggestions for useful information to add, please send an email to the developers so we can improve the DSMC documentation.

Once you are familiar with DSMC, you may want to bookmark this page at Section_commands.html#comm since it gives quick access to documentation for all DSMC commands.

PDF file of the entire manual, generated by htmldoc

1. Introduction

These sections provide an overview of what DSMC can and can't do, describe what it means for DSMC to be an open−source code, and acknowledge the funding and people who have contributed to DSMC over the years.

1.1 What is DSMC

DSMC is a classical molecular dynamics code that models an ensemble of particles in a liquid, solid, or gaseous state. It can model atomic, polymeric, biological, metallic, granular, and coarse−grained systems using a variety of force fields and boundary conditions.

For examples of DSMC simulations, see the Publications page of the DSMC WWW Site.

DSMC runs efficiently on single−processor desktop or laptop machines, but is designed for parallel computers. It will run on any parallel machine that compiles C++ and supports the MPI message−passing library. This includes distributed− or shared−memory parallel machines and Beowulf−style clusters.

DSMC can model systems with only a few particles up to millions or billions. See this section for information on DSMC performance and scalability, or the Benchmarks section of the DSMC WWW Site.

DSMC is a freely−available open−source code, distributed under the terms of the GNU Public License, which means you can use or modify the code however you wish. See this section for a brief discussion of the open−source philosophy.

DSMC is designed to be easy to modify or extend with new capabilities, such as new force fields, atom types, boundary conditions, or diagnostics. See this section for more details.

The current version of DSMC is written in C++. Earlier versions were written in F77 and F90. See this section for more information on different versions. All versions can be downloaded from the DSMC WWW Site.

DSMC was originally developed under a US Department of Energy CRADA (Cooperative Research and Development Agreement) between two DOE labs and 3 companies. It is distributed by Sandia National Labs. See this section for more information on DSMC funding and individuals who have contributed to DSMC.

In the most general sense, DSMC integrates Newton's equations of motion for collections of atoms, molecules, or macroscopic particles that interact via short− or long−range forces with a variety of initial and/or boundary conditions. For computational efficiency DSMC uses neighbor lists to keep track of nearby particles. The lists are optimized for systems with particles that are repulsive at short distances, so that the local density of particles never becomes too large. On parallel machines, DSMC uses spatial−decomposition techniques to partition the simulation domain into small 3d sub−domains, one of which is assigned to each processor. Processors communicate and store "ghost" atom information for atoms that border their sub−domain. DSMC is most efficient (in a parallel sense) for systems whose particles fill a 3d rectangular box with roughly uniform density. Papers with technical details of the algorithms used in DSMC are listed in this section.

1.2 DSMC features

This section highlights DSMC features, with pointers to specific commands which give more details. If DSMC doesn't have your favorite interatomic potential, boundary condition, or atom type, see this section, which describes how you can add it to DSMC.

General features

- runs on a single processor or in parallel
- distributed−memory message−passing parallelism (MPI)
- spatial−decomposition of simulation domain for parallelism
- open−source distribution
- highly portable C++
- optional libraries used: MPI and single−processor FFT
- easy to extend with new features and functionality
- runs from an input script
- syntax for defining and using variables and formulas
- syntax for looping over runs and breaking out of loops
- run one or multiple simulations simultaneously (in parallel) from one script
- build as library, invoke DSMC thru library interface or provided Python wrapper
- couple with other codes: DSMC calls other code, other code calls DSMC, umbrella code calls both

Particle and model types

(atom style command)

- atoms
- coarse−grained particles (e.g. bead−spring polymers)
- united−atom polymers or organic molecules
- all−atom polymers, organic molecules, proteins, DNA
- metals
- granular materials
- coarse−grained mesoscale models
- extended spherical and ellipsoidal particles
- point dipolar particles
- rigid collections of particles
- hybrid combinations of these

Force fields

(pair style, bond style, angle style, dihedral style, improper style, kspace style commands)

- pairwise potentials: Lennard−Jones, Buckingham, Morse, Born−Mayer−Huggins, Yukawa, soft, class 2 (COMPASS), hydrogen bond, tabulated
- charged pairwise potentials: Coulombic, point−dipole
- manybody potentials: EAM, Finnis/Sinclair EAM, modified EAM (MEAM), embedded ion method (EIM), EDIP, ADP, Stillinger−Weber, Tersoff, REBO, AIREBO, ReaxFF, COMB
- electron force field (eFF, AWPMD)
- coarse−grained potentials: DPD, GayBerne, REsquared, colloidal, DLVO
- mesoscopic potentials: granular, Peridynamics, SPH

- bond potentials: harmonic, FENE, Morse, nonlinear, class 2, quartic (breakable)
- angle potentials: harmonic, CHARMM, cosine, cosine/squared, cosine/periodic, class 2 (COMPASS)
- dihedral potentials: harmonic, CHARMM, multi−harmonic, helix, class 2 (COMPASS), OPLS
- improper potentials: harmonic, cvff, umbrella, class 2 (COMPASS)
- polymer potentials: all−atom, united−atom, bead−spring, breakable
- water potentials: TIP3P, TIP4P, SPC
- implicit solvent potentials: hydrodynamic lubrication, Debye
- long−range Coulombics and dispersion: Ewald, PPPM (similar to particle−mesh Ewald), Ewald/N for long−range Lennard−Jones
- force−field compatibility with common CHARMM, AMBER, DREIDING, OPLS, GROMACS, COMPASS options
- handful of GPU−enabled pair styles

hybrid potentials: multiple pair, bond, angle, dihedral, improper potentials can be used in one simulation overlaid potentials: superposition of multiple pair potentials

Atom creation

(read_data, lattice, create_atoms, delete_atoms, displace_atoms, replicate commands)

- read in atom coords from files
- create atoms on one or more lattices (e.g. grain boundaries)
- delete geometric or logical groups of atoms (e.g. voids)
- replicate existing atoms multiple times
- displace atoms

Ensembles, constraints, and boundary conditions

(fix command)

- 2d or 3d systems
- orthogonal or non−orthogonal (triclinic symmetry) simulation domains
- constant NVE, NVT, NPT, NPH, Parinello/Rahman integrators
- thermostatting options for groups and geometric regions of atoms
- pressure control via Nose/Hoover or Berendsen barostatting in 1 to 3 dimensions
- simulation box deformation (tensile and shear)
- harmonic (umbrella) constraint forces
- rigid body constraints
- SHAKE bond and angle constraints
- bond breaking, formation, swapping
- walls of various kinds
- non−equilibrium molecular dynamics (NEMD)
- variety of additional boundary conditions and constraints

Integrators

(run, run_style, minimize commands)

- velocity−Verlet integrator
- Brownian dynamics
- rigid body integration
- energy minimization via conjugate gradient or steepest descent relaxation

- rRESPA hierarchical timestepping

Diagnostics

- see the various flavors of the fix and compute commands

Output

(dump, restart commands)

- log file of thermodynamic info
- text dump files of atom coords, velocities, other per−atom quantities
- binary restart files
- parallel I/O of dump and restart files
- per−atom quantities (energy, stress, centro−symmetry parameter, CNA, etc)
- user−defined system−wide (log file) or per−atom (dump file) calculations
- spatial and time averaging of per−atom quantities
- time averaging of system−wide quantities
- atom snapshots in native, XYZ, XTC, DCD, CFG formats

Multi−replica models

nudged elastic band parallel replica dynamics temperature accelerated dynamics parallel tempering

Pre− and post−processing

- Various pre− and post−processing serial tools are packaged with DSMC; see these doc pages.
- Our group has also written and released a separate toolkit called Pizza.py which provides tools for doing setup, analysis, plotting, and visualization for DSMC simulations. Pizza.py is written in Python and is available for download from the Pizza.py WWW site.

Specialized features

These are DSMC capabilities which you may not think of as typical molecular dynamics options:

- stochastic rotation dynamics (SRD)
- real−time visualization and interactive MD
- atom−to−continuum coupling with finite elements
- coupled rigid body integration via the POEMS library
- grand canonical Monte Carlo insertions/deletions
- Direct Simulation Monte Carlo for low−density fluids
- Peridynamics mesoscale modeling
- targeted and steered molecular dynamics
- two−temperature electron model

DSMCMMPS non−features

DSMC is designed to efficiently compute Newton's equations of motion for a system of interacting particles. Many of the tools needed to pre− and post−process the data for such simulations are not included in the DSMC kernel for several reasons:

- the desire to keep DSMC simple
- they are not parallel operations
- other codes already do them
- limited development resources

Specifically, DSMC itself does not:

- run thru a GUI
- build molecular systems
- assign force−field coefficients automagically
- perform sophisticated analyses of your MD simulation
- visualize your MD simulation
- plot your output data

A few tools for pre− and post−processing tasks are provided as part of the DSMC package; they are described in this section. However, many people use other codes or write their own tools for these tasks.

As noted above, our group has also written and released a separate toolkit called Pizza.py which addresses some of the listed bullets. It provides tools for doing setup, analysis, plotting, and visualization for DSMC simulations. Pizza.py is written in Python and is available for download from the Pizza.py WWW site.

DSMC requires as input a list of initial atom coordinates and types, molecular topology information, and force−field coefficients assigned to all atoms and bonds. DSMC will not build molecular systems and assign force−field parameters for you.

For atomic systems DSMC provides a create_atoms command which places atoms on solid−state lattices (fcc, bcc, user−defined, etc). Assigning small numbers of force field coefficients can be done via the pair coeff, bond coeff, angle coeff, etc commands. For molecular systems or more complicated simulation geometries, users typically use another code as a builder and convert its output to DSMC input format, or write their own code to generate atom coordinate and molecular topology for DSMC to read in.

For complicated molecular systems (e.g. a protein), a multitude of topology information and hundreds of force−field coefficients must typically be specified. We suggest you use a program like CHARMM or AMBER or other molecular builders to setup such problems and dump its information to a file. You can then reformat the file as DSMC input. Some of the tools in this section can assist in this process.

Similarly, DSMC creates output files in a simple format. Most users post−process these files with their own analysis tools or re−format them for input into other programs, including visualization packages. If you are convinced you need to compute something on−the−fly as DSMC runs, see this section for a discussion of how you can use the dump and compute and fix commands to print out data of your choosing. Keep in mind that complicated computations can slow down the molecular dynamics timestepping, particularly if the computations are not parallel, so it is often better to leave such analysis to post−processing codes.

A very simple (yet fast) visualizer is provided with the DSMC package − see the xmovie tool in this section. It creates xyz projection views of atomic coordinates and animates them. We find it very useful for debugging purposes. For high−quality visualization we recommend the following packages:

- VMD
- AtomEye
- PyMol
- Raster3d
- RasMol

Other features that DSMC does not yet (and may never) support are discussed in this section.

Finally, these are freely−available molecular dynamics codes, most of them parallel, which may be well−suited to the problems you want to model. They can also be used in conjunction with DSMC to perform complementary modeling tasks.

- CHARMM
- AMBER
- NAMD
- NWCHEM
- DL_POLY
- Tinker

CHARMM, AMBER, NAMD, NWCHEM, and Tinker are designed primarily for modeling biological molecules. CHARMM and AMBER use atom−decomposition (replicated−data) strategies for parallelism; NAMD and NWCHEM use spatial−decomposition approaches, similar to DSMC. Tinker is a serial code. DL_POLY includes potentials for a variety of biological and non−biological materials; both a replicated−data and spatial−decomposition version exist.

1.4 Open source distribution

DSMC comes with no warranty of any kind. As each source file states in its header, it is a copyrighted code that is distributed free−of− charge, under the terms of the GNU Public License (GPL). This is often referred to as open−source distribution − see www.gnu.org or www.opensource.org for more details. The legal text of the GPL is in the LICENSE file that is included in the DSMC distribution.

Here is a summary of what the GPL means for DSMC users:

(1) Anyone is free to use, modify, or extend DSMC in any way they choose, including for commercial purposes.

(2) If you distribute a modified version of DSMC, it must remain open−source, meaning you distribute it under the terms of the GPL. You should clearly annotate such a code as a derivative version of DSMC.

(3) If you release any code that includes DSMC source code, then it must also be open−sourced, meaning you distribute it under the terms of the GPL.

(4) If you give DSMC files to someone else, the GPL LICENSE file and source file headers (including the copyright and GPL notices) should remain part of the code.

In the spirit of an open−source code, these are various ways you can contribute to making DSMC better. You can send email to the developers on any of these items.

- Point prospective users to the DSMC WWW Site. Mention it in talks or link to it from your WWW site.
- If you find an error or omission in this manual or on the DSMC WWW Site, or have a suggestion for something to clarify or include, send an email to the developers.
- If you find a bug, this section describes how to report it.
- If you publish a paper using DSMC results, send the citation (and any cool pictures or movies if you like) to add to the Publications, Pictures, and Movies pages of the DSMC WWW Site, with links and attributions back to you.
- Create a new Makefile.machine that can be added to the src/MAKE directory.
- The tools sub−directory of the DSMC distribution has various stand−alone codes for pre− and

post−processing of DSMC data. More details are given in this section. If you write a new tool that users will find useful, it can be added to the DSMC distribution.
- DSMC is designed to be easy to extend with new code for features like potentials, boundary conditions, diagnostic computations, etc. This section gives details. If you add a feature of general interest, it can be added to the DSMC distribution.
- The Benchmark page of the DSMC WWW Site lists DSMC performance on various platforms. The files needed to run the benchmarks are part of the DSMC distribution. If your machine is sufficiently different from those listed, your timing data can be added to the page.
- You can send feedback for the User Comments page of the DSMC WWW Site. It might be added to the page. No promises.
- Cash. Small denominations, unmarked bills preferred. Paper sack OK. Leave on desk. VISA also accepted. Chocolate chip cookies encouraged.

1.5 Acknowledgments and citations

DSMC development has been funded by the US Department of Energy (DOE), through its CRADA, LDRD, ASCI, and Genomes−to−Life programs and its OASCR and OBER offices.

Specifically, work on the latest version was funded in part by the US Department of Energy's Genomics:GTL program (www.doegenomestolife.org) under the project, "Carbon Sequestration in Synechococcus Sp.: From Molecular Machines to Hierarchical Modeling".

The following paper describe the basic parallel algorithms used in DSMC. If you use DSMC results in your published work, please cite DSMCaper and include a pointer to the DSMC WWW Site (http://lammps.sandia.gov):

S. J. Plimpton, **Fast Parallel Algorithms for Short−Range Molecular Dynamics**, J Comp Phys, 117, 1−19 (1995).

Other papers describing specific algorithms used in DSMC are listed under the Citing DSMC link of the DSMC WWW page.

The Publications link on the DSMC WWW page lists papers that have cited DSMC. If your paper is not listed there for some reason, feel free to send us the info. If the simulations in your paper produced cool pictures or animations, we'll be pleased to add them to the Pictures or Movies pages of the DSMC WWW site.

The core group of DSMC developers is at Sandia National Labs:

- Steve Plimpton, sjplimp at sandia.gov
- Aidan Thompson, athomps at sandia.gov
- Paul Crozier, pscrozi at sandia.gov

The following folks are responsible for significant contributions to the code, or other aspects of the DSMC development effort. Many of the packages they have written are somewhat unique to DSMC and the code would not be as general−purpose as it is without their expertise and efforts.

- Axel Kohlmeyer (Temple U), akohlmey at gmail.com, SVN and Git repositories, indefatigable mail list responder, USER−CG−CMM and USER−OMP packages
- Roy Pollock (LLNL), Ewald and PPPM solvers
- Mike Brown (ORNL), brownw at ornl.gov, GPU package
- Greg Wagner (Sandia), gjwagne at sandia.gov, MEAM package for MEAM potential
- Mike Parks (Sandia), mlparks at sandia.gov, PERI package for Peridynamics

- Rudra Mukherjee (JPL), Rudranarayan.M.Mukherjee at jpl.nasa.gov, POEMS package for articulated rigid body motion
- Reese Jones (Sandia) and collaborators, rjones at sandia.gov, USER−ATC package for atom/continuum coupling
- Ilya Valuev (JIHT), valuev at physik.hu−berlin.de, USER−AWPMD package for wave−packet MD
- Christian Trott (U Tech Ilmenau), christian.trott at tu−ilmenau.de, USER−CUDA package
- Andres Jaramillo−Botero (Caltech), ajaramil at wag.caltech.edu, USER−EFF package for electron force field
- Pieter in' t Veld (BASF), pieter.intveld at basf.com, USER−EWALDN package for 1/r^N long−range solvers
- Christoph Kloss (JKU), Christoph.Kloss at jku.at, USER−LIGGGHTS package for granular models and granular/fluid coupling
- Metin Aktulga (LBL), hmaktulga at lbl.gov, USER−REAXC package for C version of ReaxFF
- Georg Gunzenmuller (EMI), georg.ganzenmueller at emi.fhg.de, USER−SPH package

As discussed in this section, DSMC originated as a cooperative project between DOE labs and industrial partners. Folks involved in the design and testing of the original version of DSMC were the following:

- John Carpenter (Mayo Clinic, formerly at Cray Research)
- Terry Stouch (Lexicon Pharmaceuticals, formerly at Bristol Myers Squibb)
- Steve Lustig (Dupont)
- Jim Belak (LLNL)

2. Getting Started

This section describes how to build and run DSMC, for both new and experienced users.

2.1 What's in the DSMC distribution

When you download DSMC you will need to unzip and untar the downloaded file with the following commands, after placing the file in an appropriate directory.

```
gunzip lammps*.tar.gz
tar xvf lammps*.tar
```

This will create a DSMC directory containing two files and several sub−directories:

README	text file
LICENSE	the GNU General Public License (GPL)
bench	benchmark problems
couple	code coupling examples, using DSMC as a library
doc	documentation
examples	simple test problems
potentials	embedded atom method (EAM) potential files
src	source files
tools	pre− and post−processing tools

If you download one of the Windows executables from the download page, then you just get a single file:

```
lmp_windows.exe
```

Skip to the Running DSMC sections for info on how to launch these executables on a Windows box.

The Windows executables for serial or parallel only include certain packages and bug−fixes/upgrades listed on this page up to a certain date, as stated on the download page. If you want something with more packages or that is more current, you'll have to download the source tarball and build it yourself from source code using Microsoft Visual Studio, as described in the next section.

2.2 Making DSMC

This section has the following sub−sections:

- Read this first

Read this first:

Building DSMC can be non−trivial. You may need to edit a makefile, there are compiler options to consider, additional libraries can be used (MPI, FFT, JPEG), DSMC packages may be included or excluded, some of these packages use auxiliary libraries which need to be pre−built, etc.

Please read this section carefully. If you are not comfortable with makefiles, or building codes on a Unix platform, or running an MPI job on your machine, please find a local expert to help you. Many compiling, linking, and run problems that users have are not really DSMC issues − they are peculiar to the user's system, compilers, libraries, etc. Such questions are better answered by a local expert.

If you have a build problem that you are convinced is a DSMC issue (e.g. the compiler complains about a line of DSMC source code), then please send an email to the developers.

If you succeed in building DSMC on a new kind of machine, for which there isn't a similar Makefile for in the src/MAKE directory, send it to the developers and we'll include it in future DSMC releases.

Steps to build a DSMC executable:

Step 0

The src directory contains the C++ source and header files for DSMC. It also contains a top−level Makefile and a MAKE sub−directory with low−level Makefile.* files for many machines. From within the src directory, type "make" or "gmake". You should see a list of available choices. If one of those is the machine and options you want, you can type a command like:

```
make linux
or
gmake mac
```

Note that on a multi−processor or multi−core platform you can launch a parallel make, by using the "−j" switch with the make command, which will build DSMC more quickly.

If you get no errors and an executable like lmp_linux or lmp_mac is produced, you're done; it's your lucky day.

Note that by default only a few of DSMC optional pacakges are installed. To build DSMC with optional packages, see this section below.

Step 1

If Step 0 did not work, you will need to create a low−level Makefile for your machine, like Makefile.foo. You should make a copy of an existing src/MAKE/Makefile.* as a starting point. The only portions of the file you need to edit are the first line, the "compiler/linker settings" section, and the "DSMC−specific settings" section.

Step 2

Change the first line of src/MAKE/Makefile.foo to list the word "foo" after the "#", and whatever other options it will set. This is the line you will see if you just type "make".

Step 3

The "compiler/linker settings" section lists compiler and linker settings for your C++ compiler, including optimization flags. You can use g++, the open−source GNU compiler, which is available on all Unix systems. You can also use mpicc which will typically be available if MPI is installed on your system, though you should check which actual compiler it wraps. Vendor compilers often produce faster code. On boxes with Intel CPUs, we suggest using the commercial Intel icc compiler, which can be downloaded from Intel's compiler site.

If building a C++ code on your machine requires additional libraries, then you should list them as part of the LIB variable.

The DEPFLAGS setting is what triggers the C++ compiler to create a dependency list for a source file. This speeds re−compilation when source (*.cpp) or header (*.h) files are edited. Some compilers do not support dependency file creation, or may use a different switch than −D. GNU g++ works with −D. If your compiler can't create dependency files, then you'll need to create a Makefile.foo patterned after Makefile.storm, which uses different rules that do not involve dependency files. Note that when you build DSMC for the first time on a new platform, a long list of *.d files will be printed out rapidly. This is not an error; it is the Makefile doing its normal creation of dependencies.

Step 4

The "system−specific settings" section has several parts. Note that if you change any −D setting in this section, you should do a full re−compile, after typing "make clean" (which will describe different clean options).

The LMP_INC variable is used to include options that turn on ifdefs within the DSMC code. The options that are currently recogized are:

- −DDSMC_GZIP
- −DDSMC_JPEG
- −DDSMC_XDR
- −DDSMC_SMALLBIG
- −DDSMC_BIGBIG
- −DDSMC_SMALLSMALL
- −DDSMC_LONGLONG_TO_LONG
- −DPACK_ARRAY
- −DPACK_POINTER
- −DPACK_MEMCPY

The read_data and dump commands will read/write gzipped files if you compile with −DDSMC_GZIP. It requires that your Unix support the "popen" command.

If you use −DDSMC_JPEG, the dump image command will be able to write out JPEG image files. If not, it will only be able to write out text−based PPM image files. For JPEG files, you must also link DSMC with a JPEG library, as described below.

If you use −DDSMC_XDR, the build will include XDR compatibility files for doing particle dumps in XTC format. This is only necessary if your platform does have its own XDR files available. See the Restrictions section of the dump command for details.

Use at most one of the −DDSMC_SMALLBIG, −DDSMC_BIGBIG, −D−DDSMC_SMALLSMALL settings. The default is −DDSMC_SMALLBIG. These refer to use of 4−byte (small) vs 8−byte (big) integers within DSMC, as described in src/lmptype.h. The only reason to use the BIGBIG setting is to enable simulation of huge molecular systems with more than 2 billion atoms. The only reason to use the SMALLSMALL setting is if your machine does not support 64−bit integers.

The −DDSMC_LONGLONG_TO_LONG setting may be needed if your system or MPI version does not recognize "long long" data types. In this case a "long" data type is likely already 64−bits, in which case this setting will convert to that data type.

Using one of the −DPACK_ARRAY, −DPACK_POINTER, and −DPACK_MEMCPY options can make for faster parallel FFTs (in the PPPM solver) on some platforms. The −DPACK_ARRAY setting is the default. See the kspace_style command for info about PPPM. See Step 6 below for info about building DSMC with an FFT library.

Step 5

The 3 MPI variables are used to specify an MPI library to build DSMC with.

DSMC want DSMC to run in parallel, you must have an MPI library installed on your platform. If you use an MPI−wrapped compiler, such as "mpicc" to build DSMC, you should be able to leave these 3 variables blank; the MPI wrapper knows where to find the needed files. If not, and MPI is installed on your system in the usual place (under /usr/local), you also may not need to specify these 3 variables. On some large parallel machines which use "modules" for their compile/link environements, you may simply need to include the correct module in your build environment. Or the parallel machine may have a vendor−provided MPI which the compiler has no trouble finding.

Failing this, with these 3 variables you can specify where the mpi.h file (MPI_INC) and the MPI library file (MPI_PATH) are found and the name of the library file (MPI_LIB).

If you are installing MPI yourself, we recommend Argonne's MPICH 1.2 or 2.0 or OpenMPI. MPICH can be downloaded from the Argonne MPI site. OpenMPI can be downloaded the OpenMPI site. LAM MPI should also work. If you are running on a big parallel platform, your system people or the vendor should have already installed a version of MPI, which will be faster than MPICH or OpenMPI or LAM, so find out how to build and link with it. If you use MPICH or OpenMPI or LAM, you will have to configure and build it for your platform. The MPI configure script should have compiler options to enable you to use the same compiler you are using for the DSMC build, which can avoid problems that can arise when linking DSMC to the MPI library.

If you just want to run DSMC on a single processor, you can use the dummy MPI library provided in src/STUBS, since you don't need a true MPI library installed on your system. See the src/MAKE/Makefile.serial file for how to specify the 3 MPI variables in this case. You will also need to build the STUBS library for your platform before making DSMC itself. From the src directory, type "make stubs", or from the STUBS dir, type "make" and it should create a libmpi.a suitable for linking to DSMC. If this build fails, you will need to edit the STUBS/Makefile for your platform.

The file STUBS/mpi.cpp provides a CPU timer function called MPI_Wtime() that calls gettimeofday() . If your system doesn't support gettimeofday() , you'll need to insert code to call another timer. Note that the ANSI−standard function clock() rolls over after an hour or so, and is therefore insufficient for timing long DSMC simulations.

Step 6

The 3 FFT variables allow you to specify an FFT library which DSMC uses (for performing 1d FFTs) when running the particle−particle particle−mesh (PPPM) option for long−range Coulombics via the kspace_style command.

DSMC supports various open−source or vendor−supplied FFT libraries for this purpose. If you leave these 3 variables blank, DSMC will use the open−source KISS FFT library, which is included in the DSMC distribution. This library is portable to all platforms and for typical DSMC simulations is almost as fast as FFTW or vendor optimized libraries. If you are not including the KSPACE package in your build, you can also leave the 3 variables blank.

Otherwise, select which kinds of FFTs to use as part of the FFT_INC setting by a switch of the form −DFFT_XXX. Recommended values for XXX are: MKL, SCSL, FFTW2, and FFTW3. Legacy options are: INTEL, SGI, ACML, and T3E. For backward compatability, using −DFFT_FFTW will use the FFTW2 library. Using −DFFT_NONE will use the KISS library described above.

You may also need to set the FFT_INC, FFT_PATH, and FFT_LIB variables, so the compiler and linker can find the needed FFT header and library files. Note that on some large parallel machines which use "modules" for their compile/link environements, you may simply need to include the correct module in your build environment. Or the parallel machine may have a vendor−provided FFT library which the compiler has no trouble finding.

FFTW is a fast, portable library that should also work on any platform. You can download it from www.fftw.org. Both the legacy version 2.1.X and the newer 3.X versions are supported as −DFFT_FFTW2 or −DFFT_FFTW3. Building FFTW for your box should be as simple as ./configure; make. Note that on some platforms FFTW2 has been pre−installed, and uses renamed files indicating the precision it was compiled with, e.g. sfftw.h, or dfftw.h instead of fftw.h. In this case, you can specify an additional define variable for FFT_INC called −DFFTW2_SIZE, which will select the correct include file. In this case, for FFT_LIB you must also manually specify the correct library, namely −lsfftw or −ldfftw.

The FFT_INC variable also allows for a −DFFT_SINGLE setting that will use single−precision FFTs with PPPM, which can speed−up long−range calulations, particularly in parallel or on GPUs. Fourier transform and related PPPM operations are somewhat insensitive to floating point truncation errors and thus do not always need to be performed in double precision. Using the −DFFT_SINGLE setting trades off a little accuracy for reduced memory use and parallel communication costs for transposing 3d FFT data. Note that single precision FFTs have only been tested with the FFTW3, FFTW2, MKL, and KISS FFT options.

Step 7

The 3 JPG variables allow you to specify a JPEG library which DSMC uses when writing out JPEG files via the dump image command. These can be left blank if you do not use the −DDSMC_JPEG switch discussed above in Step 4, since in that case JPEG output will be disabled.

A standard JPEG library usually goes by the name libjpeg.a and has an associated header file jpeglib.h. Whichever JPEG library you have on your platform, you'll need to set the appropriate JPG_INC, JPG_PATH, and JPG_LIB variables, so that the compiler and linker can find it.

As before, if these header and library files are in the usual place on your machine, you may not need to set these variables.

Step 8

Note that by default only a few of DSMC optional pacakges are installed. To build DSMC with optional packages, see this section below, before proceeding to Step 9.

Step 9

That's it. Once you have a correct Makefile.foo, you have installed the optional DSMC packages you want to include in your build, and you have pre−built any other needed libraries (e.g. MPI, FFT, package libraries), all you need to do from the src directory is type something like this:

```
make foo
or
gmake foo
```

You should get the executable lmp_foo when the build is complete.

Errors that can occur when making DSMC:

IMPORTANT NOTE: If an error occurs when building DSMC, the compiler or linker will state very explicitly what the problem is. The error message should give you a hint as to which of the steps above has failed, and what you need to do in order to fix it. Building a code with a Makefile is a very logical process. The compiler and linker need to find the appropriate files and those files need to be compatible with DSMC source files. When a make fails, there is usually a very simple reason, which you or a local expert will need to fix.

Here are two non−obvious errors that can occur:

(1) If the make command breaks immediately with errors that indicate it can't find files with a "*" in their names, this can be because your machine's native make doesn't support wildcard expansion in a makefile. Try gmake instead of make. If that doesn't work, try using a −f switch with your make command to use a pre−generated Makefile.list which explicitly lists all the needed files, e.g.

```
make makelist
make −f Makefile.list linux
gmake −f Makefile.list mac
```

The first "make" command will create a current Makefile.list with all the file names in your src dir. The 2nd "make" command (make or gmake) will use it to build DSMC. Note that you should include/exclude any desired optional packages before using the "make makelist" command.

(2) If you get an error that says something like 'identifier "atoll" is undefined', then your machine does not support "long long" integers. Try using the −DDSMC_LONGLONG_TO_LONG setting described above in Step 4.

Additional build tips:

(1) Building DSMC for multiple platforms.

You can make DSMC for multiple platforms from the same src directory. Each target creates its own object sub−directory called Obj_name where it stores the system−specific *.o files.

(2) Cleaning up.

Typing "make clean−all" or "make clean−foo" will delete *.o object files created when DSMC is built, for either all builds or for a particular machine.

(3) Changing the size limits in src/lmptype.h

If you are running a very large problem (billions of atoms or more) and get a run−time error about the system being too big, either on a per−processor basis or in total size, then you may need to change one or more settings in src/lmptype.h and re−compile DSMC.

As the documentation in that file explains, you have basically two choices to make:

- set the data type size of integer atom IDs to 4 or 8 bytes
- set the data type size of integers that store the total system size to 4 or 8 bytes

The default for atom IDs is 4−byte integers since there is a memory and communication cost for 8−byte integers. Non−molecular problems do not need atom IDs so this does not restrict their size. Molecular problems (which use IDs to define molecular topology), are limited to about 2 billion atoms (2^31) with 4−byte IDs. With 8−byte IDs they are effectively unlimited in size (2^63).

The default for total system size quantities (like the number of atoms or timesteps) is 8−byte integers by default which is effectively unlimited in size (2^63). If your system or MPI implementation does not support 8−byte integers, an error will be generated, and you will need to set "bigint" to 4−byte integers. This restricts your total system size to about 2 billion atoms or timesteps (2^31).

Note that in src/lmptype.h there are also settings for the MPI data types associated with the integers that store atom IDs and total system sizes, which need to be set consistent with the associated C data types.

In all cases, the size of problem that can be run on a per−processor basis is limited by 4−byte integer storage to about 2 billion atoms per processor (2^31), which should not normally be a restriction since such a problem would have a huge per−processor memory footprint due to neighbor lists and would run very slowly in terms of CPU secs/timestep.

Building for a Mac:

OS X is BSD Unix, so it should just work. See the Makefile.mac file.

Building for Windows:

The DSMC download page has an option to download both a serial and parallel pre−built Windows exeutable. See the Running DSMC section for instructions for running these executables on a Windows box.

If the pre−built executable doesn't have the options you want, then you can build DSMC from its source files on a Windows box. One way to do this is install and use cygwin to build DSMC with a standard Linus make, just as you would on any Linux box; see src/MAKE/Makefile.cygwin.

There is a also a src/WINDOWS directory that contains project files for Microsoft Visual Studio 2005, which should also work with later versions of VS. That directory contains a README.txt file which provides instructions for building DSMC from source code using Visual Studio that are hopefully easy to follow for Windows and VS users.

Four VS project options are provided. The first includes the default packages (MANYBODY, MOLECULE, and KSPACE). The second includes all standard packages (except GPU, MEAM, and REAX which are not yet included because they require NVIDIA or Fortran compilation). The third includes all standard packages (with the exceptions) and some user packages. The included user packages are USER−EFF, USER−CG−CMM, and USER−REAXC. The fourth project includes the USER−AWPMD package.

2.3 Making DSMC with optional packages

This section has the following sub−sections:

- Package basics
- Including/excluding packages
- Packages that require extra libraries
- Additional Makefile settings for extra libraries

Package basics:

The source code for DSMC is structured as a set of core files which are always included, plus optional packages. Packages are groups of files that enable a specific set of features. For example, force fields for molecular systems or granular systems are in packages. You can see the list of all packages by typing "make package" from within the src directory of the DSMC distribution.

If you use a command in a DSMC input script that is specific to a particular package, you must have built DSMC with that package, else you will get an error that the style is invalid or the command is unknown. Every command's doc page specfies if it is part of a package. You can also type

```
lmp_machine −h
```

to run your executable with the optional −h command−line switch for "help", which will list the styles and commands known to your executable.

There are two kinds of packages in DSMC, standard and user packages. More information about the contents of standard and user packages is given in this section of the manual. The difference between standard and user packages is as follows:

Standard packages are supported by the DSMC developers and are written in a syntax and style consistent with the rest of DSMC. This means we will answer questions about them, debug and fix them if necessary, and keep them compatible with future changes to DSMC.

User packages have been contributed by users, and always begin with the user prefix. If they are a single command (single file), they are typically in the user−misc package. Otherwise, they are a a set of files grouped together which add a specific functionality to the code.

User packages don't necessarily meet the requirements of the standard packages. If you have problems using a feature provided in a user package, you will likely need to contact the contributor directly to get help. Information on how to submit additions you make to DSMC as a user−contributed package is given in this section of the documentation.

Including/excluding packages:

To use or not use a package you must include or exclude it before building DSMC. From the src directory, this is typically as simple as:

```
make yes−colloid
make g++
```

or

```
make no−manybody
```

```
make g++
```

Some packages have individual files that depend on other packages being included. DSMC checks for this and does the right thing. I.e. individual files are only included if their dependencies are already included. Likewise, if a package is excluded, other files dependent on that package are also excluded.

The reason to exclude packages is if you will never run certain kinds of simulations. For some packages, this will keep you from having to build auxiliary libraries (see below), and will also produce a smaller executable which may run a bit faster.

When you download a DSMC tarball, these packages are pre−installed in the src directory: KSPACE, MANYBODY,MOLECULE. When you download DSMC source files from the SVN or Git repositories, no packages are pre−installed.

Packages are included or excluded by typing "make yes−name" or "make no−name", where "name" is the name of the package in lower−case, e.g. name = kspace for the KSPACE package or name = user−atc for the USER−ATC package. You can also type "make yes−standard", "make no−standard", "make yes−user", "make no−user", "make yes−all" or "make no−all" to include/exclude various sets of packages. Type "make package" to see the all of the package−related make options.

IMPORTANT NOTE: Inclusion/exclusion of a package works by simply moving files back and forth between the main src directory and sub−directories with the package name (e.g. src/KSPACE, src/USER−ATC), so that the files are seen or not seen when DSMC is built. After you have included or excluded a package, you must re−build DSMC.

Additional package−related make options exist to help manage DSMC files that exist in both the src directory and in package sub−directories. You do not normally need to use these commands unless you are editing DSMC files or have downloaded a patch from the DSMC WWW site.

Typing "make package−update" will overwrite src files with files from the package sub−directories if the package has been included. It should be used after a patch is installed, since patches only update the files in the package sub−directory, but not the src files. Typing "make package−overwrite" will overwrite files in the package sub−directories with src files.

Typing "make package−status" will show which packages are currently included. Of those that are included, it will list files that are different in the src directory and package sub−directory. Typing "make package−diff" lists all differences between these files. Again, type "make package" to see all of the package−related make options.

Packages that require extra libraries:

A few of the standard and user packages require additional auxiliary libraries to be compiled first. If you get a DSMC build error about a missing library, this is likely the reason. The source code for these libraries is included in the DSMC distribution under the "lib" directory. Look at the lib/README file for a list of these or see this section of the doc pages.

Each lib directory has a README file (e.g. lib/reax/README) with instructions on how to build that library. Typically this is done in this manner:

```
make −f Makefile.g++
```

in the appropriate directory, e.g. in lib/reax. Some of the libraries do not build this way. Again, see the libary README file for details.

In any event, you will need to use a Makefile that is a match for your system. If one of the provided Makefiles is not appropriate for your system you will need to edit or add one. For example, in the case of Fortran−based libraries, your system must have a Fortran compiler, the settings for which will need to be listed in the Makefile.

When you have built one of these libraries, there are 2 things to check:

(1) The file libname.a should now exist in lib/name. E.g. lib/reax/libreax.a. This is the library file DSMC will link against. One exception is the lib/cuda library which produces the file liblammpscuda.a, because there is already a system library libcuda.a.

(2) The file Makefile.lammps should exist in lib/name. E.g. lib/cuda/Makefile.lammps. This file may be auto−generated by the build of the library, or you may need to make a copy of the appropriate provided file (e.g. lib/meam/Makefile.lammps.gfortran). Either way you should insure that the settings in this file are appropriate for your system.

There are typically 3 settings in the Makefile.lammps file (unless some are blank or not needed): a SYSINC, SYSPATH, and SYSLIB setting, specific to this package. These are settings the DSMC build will import when compiling the DSMC package files (not the library files), and linking to the auxiliary library. They typically list any other system libraries needed to support the package and where to find them. An example is the BLAS and LAPACK libraries needed by the USER−ATC package. Or the system libraries that support calling Fortran from C++, as the MEAM and REAX packages do.

Note that if these settings are not correct for your box, the DSMC build will likely fail.

2.4 Building DSMC as a library

DSMC itself can be built as a library, which can then be called from another application or a scripting language. See this section for more info on coupling DSMC to other codes. Building DSMC as a library is done by typing

```
make makelib
make -f Makefile.lib foo
```

where foo is the machine name. Note that inclusion or exclusion of any desired optional packages should be done before typing "make makelib". The first "make" command will create a current Makefile.lib with all the file names in your src dir. The 2nd "make" command will use it to build DSMC as a library. This requires that Makefile.foo have a library target (lib) and system−specific settings for ARCHIVE and ARFLAGS. See Makefile.linux for an example. The build will create the file liblmp_foo.a which another application can link to.

When used from a C++ program, the library allows one or more DSMC objects to be instantiated. All of DSMC is wrapped in a DSMC_NS namespace; you can safely use any of its classes and methods from within your application code, as needed.

When used from a C or Fortran program or a scripting language, the library has a simple function−style interface, provided in src/library.cpp and src/library.h.

See the sample codes couple/simple/simple.cpp and simple.c as examples of C++ and C codes that invoke DSMC thru its library interface. There are other examples as well in the couple directory which are discussed in this section of the manual. See this section of the manual for a description of the Python wrapper provided with DSMC that operates through the DSMC library interface.

The files src/library.cpp and library.h contain the C−style interface to DSMC. See this section of the manual for a description of the interface and how to extend it for your needs.

2.5 Running DSMC

By default, DSMC runs by reading commands from stdin; e.g. lmp_linux < in.file. This means you first create an input script (e.g. in.file) containing the desired commands. This section describes how input scripts are structured and what commands they contain.

You can test DSMC on any of the sample inputs provided in the examples or bench directory. Input scripts are named in.* and sample outputs are named log.*.name.P where name is a machine and P is the number of processors it was run on.

Here is how you might run a standard Lennard−Jones benchmark on a Linux box, using mpirun to launch a parallel job:

```
cd src
make linux
cp lmp_linux ../bench
cd ../bench
mpirun −np 4 lmp_linux <in.lj
```

See this page for timings for this and the other benchmarks on various platforms.

On a Windows box, you can skip making DSMC and simply download an executable, as described above, though the pre−packaged executables include only certain packages.

To run a DSMC executable on a Windows machine, first decide whether you want to download the non−MPI (serial) or the MPI (parallel) version of the executable. Download and save the version you have chosen.

For the non−MPI version, follow these steps:

- Get a command prompt by going to Start−>Run... , then typing "cmd".
- Move to the directory where you have saved lmp_win_no−mpi.exe (e.g. by typing: cd "Documents").
- At the command prompt, type "lmp_win_no−mpi −in in.lj", replacing in.lj with the name of your DSMC input script.

For the MPI version, which allows you to run DSMC under Windows on multiple processors, follow these steps:

- Download and install MPICH2 for Windows.
- You'll need to use the mpiexec.exe and smpd.exe files from the MPICH2 package. Put them in same directory (or path) as the DSMC Windows executable.
- Get a command prompt by going to Start−>Run... , then typing "cmd".
- Move to the directory where you have saved lmp_win_mpi.exe (e.g. by typing: cd "Documents").
- Then type something like this: "mpiexec −np 4 −localonly lmp_win_mpi −in in.lj", replacing in.lj with the name of your DSMC input script.
- Note that you may need to provide smpd with a passphrase −−− it doesn't matter what you type.
- In this mode, output may not immediately show up on the screen, so if your input script takes a long time to execute, you may need to be patient before the output shows up.
- Alternatively, you can still use this executable to run on a single processor by typing something like: "lmp_win_mpi −in in.lj".

The screen output from DSMC is described in the next section. As it runs, DSMC also writes a log.lammps file with the same information.

Note that this sequence of commands copies the DSMC executable (lmp_linux) to the directory with the input files. This may not be necessary, but some versions of MPI reset the working directory to where the executable is, rather than leave it as the directory where you launch mpirun from (if you launch lmp_linux on its own and not under mpirun). If that happens, DSMC will look for additional input files and write its output files to the executable directory, rather than your working directory, which is probably not what you want.

If DSMC encounters errors in the input script or while running a simulation it will print an ERROR message and stop or a WARNING message and continue. See this section for a discussion of the various kinds of errors DSMC can or can't detect, a list of all ERROR and WARNING messages, and what to do about them.

DSMC can run a problem on any number of processors, including a single processor. In theory you should get identical answers on any number of processors and on any machine. In practice, numerical round−off can cause slight differences and eventual divergence of molecular dynamics phase space trajectories.

DSMC can run as large a problem as will fit in the physical memory of one or more processors. If you run out of memory, you must run on more processors or setup a smaller problem.

2.6 Command−line options

At run time, DSMC recognizes several optional command−line switches which may be used in any order. Either the full word or a one−or−two letter abbreviation can be used:

- −c or −cuda
- −e or −echo
- −i or −in
- −h or −help
- −l or −log
- −p or −partition
- −pl or −plog
- −ps or −pscreen
- −sc or −screen
- −sf or −suffix
- −v or −var

For example, lmp_ibm might be launched as follows:

```
mpirun −np 16 lmp_ibm −v f tmp.out −l my.log −sc none <in.alloy
mpirun −np 16 lmp_ibm −var f tmp.out −log my.log −screen none  <in.alloy
```

Here are the details on the options:

```
−cuda on/off
```

Explicitly enable or disable CUDA support, as provided by the USER−CUDA package. If DSMC is built with this package, as described above in Section 2.3, then by default DSMC will run in CUDA mode. If this switch is set to "off", then it will not, even if it was built with the USER−CUDA package, which means you can run standard DSMC or with the GPU package for testing or benchmarking purposes. The only reason to set the switch to "on", is to check if DSMC was built with the USER−CUDA package, since an error will be generated if it was not.

```
−echo style
```

Set the style of command echoing. The style can be *none* or *screen* or *log* or *both*. Depending on the style, each command read from the input script will be echoed to the screen and/or logfile. This can be useful to figure out which line of your script is causing an input error. The default value is *log*. The echo style can also be set by using the echo command in the input script itself.

`-in file`

Specify a file to use as an input script. This is an optional switch when running DSMC in one−partition mode. If it is not specified, DSMC reads its input script from stdin − e.g. lmp_linux < in.run. This is a required switch when running DSMC in multi−partition mode, since multiple processors cannot all read from stdin.

`-help`

Print a list of options compiled into this executable for each DSMC style (atom_style, fix, compute, pair_style, bond_style, etc). This can help you know if the command you want to use was included via the appropriate package. DSMC will print the info and immediately exit if this switch is used.

`-log file`

Specify a log file for DSMC to write status information to. In one−partition mode, if the switch is not used, DSMC writes to the file log.lammps. If this switch is used, DSMC writes to the specified file. In multi−partition mode, if the switch is not used, a log.lammps file is created with hi−level status information. Each partition also writes to a log.lammps.N file where N is the partition ID. If the switch is specified in multi−partition mode, the hi−level logfile is named "file" and each partition also logs information to a file.N. For both one−partition and multi−partition mode, if the specified file is "none", then no log files are created. Using a log command in the input script will override this setting. Option −plog will override the name of the partition log files file.N.

`-partition 8x2 4 5 ...`

Invoke DSMC in multi−partition mode. When DSMC is run on P processors and this switch is not used, DSMC runs in one partition, i.e. all P processors run a single simulation. If this switch is used, the P processors are split into separate partitions and each partition runs its own simulation. The arguments to the switch specify the number of processors in each partition. Arguments of the form MxN mean M partitions, each with N processors. Arguments of the form N mean a single partition with N processors. The sum of processors in all partitions must equal P. Thus the command "−partition 8x2 4 5" has 10 partitions and runs on a total of 25 processors.

Note that with MPI installed on a machine (e.g. your desktop), you can run on more (virtual) processors than you have physical processors. This can be useful for running multi−replica simulations, on one or a few processors.

The input script specifies what simulation is run on which partition; see the variable and next commands. This howto section gives examples of how to use these commands in this way. Simulations running on different partitions can also communicate with each other; see the temper command.

`-plog file`

Specify the base name for the partition log files, so partition N writes log information to file.N. If file is none, then no partition log files are created. This overrides the filename specified in the −log command−line option. This option is useful when working with large numbers of partitions, allowing the partition log files to be suppressed (−plog none) or placed in a sub−directory (−plog replica_files/log.lammps) If this option is not used the log file for partition N is log.lammps.N or whatever is specified by the −log command−line option.

`-pscreen file`

Specify the base name for the partition screen file, so partition N writes screen information to file.N. If file is none, then no partition screen files are created. This overrides the filename specified in the −screen command−line option. This option is useful when working with large numbers of partitions, allowing the partition screen files to be suppressed (−pscreen none) or placed in a sub−directory (−pscreen replica_files/screen) If this option is not used the screen file for partition N is screen.N or whatever is specified by the −screen command−line option.

```
−screen file
```

Specify a file for DSMC to write its screen information to. In one−partition mode, if the switch is not used, DSMC writes to the screen. If this switch is used, DSMC writes to the specified file instead and you will see no screen output. In multi−partition mode, if the switch is not used, hi−level status information is written to the screen. Each partition also writes to a screen.N file where N is the partition ID. If the switch is specified in multi−partition mode, the hi−level screen dump is named "file" and each partition also writes screen information to a file.N. For both one−partition and multi−partition mode, if the specified file is "none", then no screen output is performed. Option −pscreen will override the name of the partition screen files file.N.

```
−suffix style
```

Use variants of various styles if they exist. The specified style can be *opt* or *gpu* or *cuda*. These refer to optional packages that DSMC can be built with, as described above in Section 2.3. The "opt" style corrsponds to the OPT package, the "gpu" style to the GPU package, and the "cuda" style to the USER−CUDA package.

As an example, all of the packages provide a pair_style lj/cut variant, with style names lj/cut/opt or lj/cut/gpu or lj/cut/cuda. A variant styles can be specified explicitly in your input script, e.g. pair_style lj/cut/gpu. If the −suffix switch is used, you do not need to modify your input script. The specified suffix (opt,gpu,cuda) is automatically appended whenever your input script command creates a new atom, pair, fix, compute, or run style. atom, pair, fix, compute, or integrate style. If the variant version does not exist, the standard version is created.

The suffix command can also set a suffix and it can also turn off/on any suffix setting made via the command line.

```
−var name value1 value2 ...
```

Specify a variable that will be defined for substitution purposes when the input script is read. "Name" is the variable name which can be a single character (referenced as $x in the input script) or a full string (referenced as ${abc}). An index−style variable will be created and populated with the subsequent values, e.g. a set of filenames. Using this command−line option is equivalent to putting the line "variable name index value1 value2 ..." at the beginning of the input script. Defining an index variable as a command−line argument overrides any setting for the same index variable in the input script, since index variables cannot be re−defined. See the variable command for more info on defining index and other kinds of variables and this section for more info on using variables in input scripts.

2.7 DSMC screen output

As DSMC reads an input script, it prints information to both the screen and a log file about significant actions it takes to setup a simulation. When the simulation is ready to begin, DSMC performs various initializations and prints the amount of memory (in MBytes per processor) that the simulation requires. It also prints details of the initial thermodynamic state of the system. During the run itself, thermodynamic information is printed periodically, every few timesteps. When the run concludes, DSMC prints the final thermodynamic state and a total run time for the simulation. It then appends statistics about the CPU time and storage requirements for the simulation. An example set of statistics is shown here:

```
Loop time of 49.002 on 2 procs for 2004 atoms
```

```
Pair   time (%) = 35.0495 (71.5267)
Bond   time (%) = 0.092046 (0.187841)
Kspce  time (%) = 6.42073 (13.103)
Neigh  time (%) = 2.73485 (5.5811)
Comm   time (%) = 1.50291 (3.06703)
Outpt  time (%) = 0.013799 (0.0281601)
Other  time (%) = 2.13669 (4.36041)

Nlocal:    1002 ave, 1015 max, 989 min
Histogram: 1 0 0 0 0 0 0 0 0 1
Nghost:    8720 ave, 8724 max, 8716 min
Histogram: 1 0 0 0 0 0 0 0 0 1
Neighs:    354141 ave, 361422 max, 346860 min
Histogram: 1 0 0 0 0 0 0 0 0 1

Total # of neighbors = 708282
Ave neighs/atom = 353.434
Ave special neighs/atom = 2.34032
Number of reneighborings = 42
Dangerous reneighborings = 2
```

The first section gives the breakdown of the CPU run time (in seconds) into major categories. The second section lists the number of owned atoms (Nlocal), ghost atoms (Nghost), and pair−wise neighbors stored per processor. The max and min values give the spread of these values across processors with a 10−bin histogram showing the distribution. The total number of histogram counts is equal to the number of processors.

The last section gives aggregate statistics for pair−wise neighbors and special neighbors that DSMC keeps track of (see the special_bonds command). The number of times neighbor lists were rebuilt during the run is given as well as the number of potentially "dangerous" rebuilds. 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 insure no force interactions are missed by atoms moving beyond the neighbor skin distance before a rebuild takes place.

If an energy minimization was performed via the minimize command, additional information is printed, e.g.

```
Minimization stats:
  E initial, next-to-last, final = -0.895962 -2.94193 -2.94342
  Gradient 2-norm init/final= 1920.78 20.9992
  Gradient inf-norm init/final= 304.283 9.61216
  Iterations = 36
  Force evaluations = 177
```

The first line lists the initial and final energy, as well as the energy on the next−to−last iteration. The next 2 lines give a measure of the gradient of the energy (force on all atoms). The 2−norm is the "length" of this force vector; the inf−norm is the largest component. The last 2 lines are statistics on how many iterations and force−evaluations the minimizer required. Multiple force evaluations are typically done at each iteration to perform a 1d line minimization in the search direction.

If a kspace_style long−range Coulombics solve was performed during the run (PPPM, Ewald), then additional information is printed, e.g.

```
FFT time (% of Kspce) = 0.200313 (8.34477)
FFT Gflps 3d 1d-only = 2.31074 9.19989
```

The first line gives the time spent doing 3d FFTs (4 per timestep) and the fraction it represents of the total KSpace time (listed above). Each 3d FFT requires computation (3 sets of 1d FFTs) and communication (transposes). The total flops performed is 5Nlog_2(N), where N is the number of points in the 3d grid. The FFTs are timed with and

without the communication and a Gflop rate is computed. The 3d rate is with communication; the 1d rate is without (just the 1d FFTs). Thus you can estimate what fraction of your FFT time was spent in communication, roughly 75% in the example above.

2.8 Tips for users of previous DSMC versions

The current C++ began with a complete rewrite of DSMC 2001, which was written in F90. Features of earlier versions of DSMC are listed in this section. The F90 and F77 versions (2001 and 99) are also freely distributed as open−source codes; check the DSMC WWW Site for distribution information if you prefer those versions. The 99 and 2001 versions are no longer under active development; they do not have all the features of C++ DSMC.

If you are a previous user of DSMC 2001, these are the most significant changes you will notice in C++ DSMC:

(1) The names and arguments of many input script commands have changed. All commands are now a single word (e.g. read_data instead of read data).

(2) All the functionality of DSMC 2001 is included in C++ DSMC, but you may need to specify the relevant commands in different ways.

(3) The format of the data file can be streamlined for some problems. See the read_data command for details. The data file section "Nonbond Coeff" has been renamed to "Pair Coeff" in C++ DSMC.

(4) Binary restart files written by DSMC 2001 cannot be read by C++ DSMC with a read_restart command. This is because they were output by F90 which writes in a different binary format than C or C++ writes or reads. Use the *restart2data* tool provided with DSMC 2001 to convert the 2001 restart file to a text data file. Then edit the data file as necessary before using the C++ DSMC read_data command to read it in.

(5) There are numerous small numerical changes in C++ DSMC that mean you will not get identical answers when comparing to a 2001 run. However, your initial thermodynamic energy and MD trajectory should be close if you have setup the problem for both codes the same.

3. Commands

This section describes how a DSMC input script is formatted and what commands are used to define a DSMC simulation.

3.1 DSMC input script

DSMC executes by reading commands from a input script (text file), one line at a time. When the input script ends, DSMC exits. Each command causes DSMC to take some action. It may set an internal variable, read in a file, or run a simulation. Most commands have default settings, which means you only need to use the command if you wish to change the default.

In many cases, the ordering of commands in an input script is not important. However the following rules apply:

(1) DSMC does not read your entire input script and then perform a simulation with all the settings. Rather, the input script is read one line at a time and each command takes effect when it is read. Thus this sequence of commands:

```
timestep 0.5
run      100
run      100
```

does something different than this sequence:

```
run      100
timestep 0.5
run      100
```

In the first case, the specified timestep (0.5 fmsec) is used for two simulations of 100 timesteps each. In the 2nd case, the default timestep (1.0 fmsec) is used for the 1st 100 step simulation and a 0.5 fmsec timestep is used for the 2nd one.

(2) Some commands are only valid when they follow other commands. For example you cannot set the temperature of a group of atoms until atoms have been defined and a group command is used to define which atoms belong to the group.

(3) Sometimes command B will use values that can be set by command A. This means command A must precede command B in the input script if it is to have the desired effect. For example, the read_data command initializes the system by setting up the simulation box and assigning atoms to processors. If default values are not desired, the processors and boundary commands need to be used before read_data to tell DSMC how to map processors to the simulation box.

Many input script errors are detected by DSMC and an ERROR or WARNING message is printed. This section gives more information on what errors mean. The documentation for each command lists restrictions on how the

command can be used.

3.2 Parsing rules

Each non−blank line in the input script is treated as a command. DSMC commands are case sensitive. Command names are lower−case, as are specified command arguments. Upper case letters may be used in file names or user−chosen ID strings.

Here is how each line in the input script is parsed by DSMC:

(1) If the last printable character on the line is a ""character (with no surrounding quotes), the command is assumed to continue on the next line. The next line is concatenated to the previous line by removing the ""character and newline. This allows long commands to be continued across two or more lines.

(2) All characters from the first "#" character onward are treated as comment and discarded. See an exception in (6). Note that a comment after a trailing ""character will prevent the command from continuing on the next line. Also note that for multi−line commands a single leading "#" will comment out the entire command.

(3) The line is searched repeatedly for $ characters, which indicate variables that are replaced with a text string. See an exception in (6). If the $ is followed by curly brackets, then the variable name is the text inside the curly brackets. If no curly brackets follow the $, then the variable name is the single character immediately following the $. Thus ${myTemp} and $x refer to variable names "myTemp" and "x". See the variable command for details of how strings are assigned to variables and how they are substituted for in input script commands.

(4) The line is broken into "words" separated by whitespace (tabs, spaces). Note that words can thus contain letters, digits, underscores, or punctuation characters.

(5) The first word is the command name. All successive words in the line are arguments.

(6) If you want text with spaces to be treated as a single argument, it can be enclosed in either double or single quotes. E.g.

```
print "Volume = $v"
print 'Volume = $v'
```

The quotes are removed when the single argument is stored internally. See the dump modify format or if commands for examples. A "#" or "$" character that is between quotes will not be treated as a comment indicator in (2) or substituted for as a variable in (3).

IMPORTANT NOTE: If the argument is itself a command that requires a quoted argument (e.g. using a print command as part of an if or run every command), then the double and single quotes can be nested in the usual manner. See the doc pages for those commands for examples. Only one of level of nesting is allowed, but that should be sufficient for most use cases.

3.3 Input script structure

This section describes the structure of a typical DSMC input script. The "examples" directory in the DSMC distribution contains many sample input scripts; the corresponding problems are discussed in this section, and animated on the DSMC WWW Site.

A DSMC input script typically has 4 parts:

1. Initialization
2. Atom definition
3. Settings
4. Run a simulation

The last 2 parts can be repeated as many times as desired. I.e. run a simulation, change some settings, run some more, etc. Each of the 4 parts is now described in more detail. Remember that almost all the commands need only be used if a non−default value is desired.

(1) Initialization

Set parameters that need to be defined before atoms are created or read−in from a file.

The relevant commands are units, dimension, newton, processors, boundary, atom_style, atom_modify.

If force−field parameters appear in the files that will be read, these commands tell DSMC what kinds of force fields are being used: pair_style, bond_style, angle_style, dihedral_style, improper_style.

(2) Atom definition

There are 3 ways to define atoms in DSMC. Read them in from a data or restart file via the read_data or read_restart commands. These files can contain molecular topology information. Or create atoms on a lattice (with no molecular topology), using these commands: lattice, region, create_box, create_atoms. The entire set of atoms can be duplicated to make a larger simulation using the replicate command.

(3) Settings

Once atoms and molecular topology are defined, a variety of settings can be specified: force field coefficients, simulation parameters, output options, etc.

Force field coefficients are set by these commands (they can also be set in the read−in files): pair_coeff, bond_coeff, angle_coeff, dihedral_coeff, improper_coeff, kspace_style, dielectric, special_bonds.

Various simulation parameters are set by these commands: neighbor, neigh_modify, group, timestep, reset_timestep, run_style, min_style, min_modify.

Fixes impose a variety of boundary conditions, time integration, and diagnostic options. The fix command comes in many flavors.

Various computations can be specified for execution during a simulation using the compute, compute_modify, and variable commands.

Output options are set by the thermo, dump, and restart commands.

(4) Run a simulation

A molecular dynamics simulation is run using the run command. Energy minimization (molecular statics) is performed using the minimize command. A parallel tempering (replica−exchange) simulation can be run using the temper command.

3.4 Commands listed by category

This section lists all DSMC commands, grouped by category. The next section lists the same commands alphabetically. Note that some style options for some commands are part of specific DSMC packages, which means they cannot be used unless the package was included when DSMC was built. Not all packages are included in a default DSMC build. These dependencies are listed as Restrictions in the command's documentation.

Initialization:

atom_modify, atom_style, boundary, dimension, newton, processors, units

Atom definition:

create_atoms, create_box, lattice, read_data, read_restart, region, replicate

Force fields:

angle_coeff, angle_style, bond_coeff, bond_style, dielectric, dihedral_coeff, dihedral_style, improper_coeff, improper_style, kspace_modify, kspace_style, pair_coeff, pair_modify, pair_style, pair_write, special_bonds

Settings:

communicate, group, mass, min_modify, min_style, neigh_modify, neighbor, reset_timestep, run_style, set, timestep, velocity

Fixes:

fix, fix_modify, unfix

Computes:

compute, compute_modify, uncompute

Output:

dump, dump image, dump_modify, restart, thermo, thermo_modify, thermo_style, undump, write_restart

Actions:

delete_atoms, delete_bonds, displace_atoms, displace_box, minimize, neb prd, run, temper

Miscellaneous:

clear, echo, if, include, jump, label, log, next, print, shell, variable

---

3.5 Individual commands

This section lists all DSMC commands alphabetically, with a separate listing below of styles within certain commands. The previous section lists the same commands, grouped by category. Note that some style options for some commands are part of specific DSMC packages, which means they cannot be used unless the package was included when DSMC was built. Not all packages are included in a default DSMC build. These dependencies are listed as Restrictions in the command's documentation.

angle_coeff	angle_style	atom_modify	atom_style	bond_coeff	bond_style
boundary	change_box	clear	communicate	compute	compute_modify
create_atoms	create_box	delete_atoms	delete_bonds	dielectric	dihedral_coeff
dihedral_style	dimension	displace_atoms	displace_box	dump	dump image
dump_modify	echo	fix	fix_modify	group	if
improper_coeff	improper_style	include	jump	kspace_modify	kspace_style
label	lattice	log	mass	minimize	min_modify
min_style	neb	neigh_modify	neighbor	newton	next
package	pair_coeff	pair_modify	pair_style	pair_write	prd
print	processors	read_data	read_restart	region	replicate
reset_timestep	restart	run	run_style	set	shell
special_bonds	suffix	tad	temper	thermo	thermo_modify
thermo_style	timestep	uncompute	undump	unfix	units
variable	velocity	write_restart			

Fix styles

See the fix command for one−line descriptions of each style or click on the style itself for a full description:

adapt	addforce	aveforce	ave/atom	ave/correlate	ave/histo	ave/spatial	ave/time
bond/break	bond/create	bond/swap	box/relax	deform	deposit	drag	dt/reset
efield	enforce2d	evaporate	external	freeze	gcmc	gravity	heat
indent	langevin	lineforce	momentum	move	msst	neb	nph
nphug	nph/asphere	nph/sphere	npt	npt/asphere	npt/sphere	nve	nve/asphere
nve/limit	nve/noforce	nve/sphere	nvt	nvt/asphere	nvt/sllod	nvt/sphere	orient/fcc
planeforce	poems	pour	press/berendsen	print	qeq/comb	reax/bonds	recenter
restrain	rigid	rigid/nve	rigid/nvt	setforce	shake	spring	spring/rg
spring/self	srd	store/force	store/state	temp/berendsen	temp/rescale	thermal/conductivity	tmd
ttm	viscosity	viscous	wall/colloid	wall/gran	wall/harmonic	wall/lj126	wall/lj93
wall/reflect	wall/region	wall/srd					

These are fix styles contributed by users, which can be used if DSMC is built with the appropriate package.

atc	imd	langevin/eff	meso	meso/stationary	nph/eff
npt/eff	nve/eff	nvt/eff	nvt/sllod/eff	qeq/reax	smd
temp/rescale/eff					

These are accelerated fix styles, which can be used if DSMC is built with the appropriate accelerated package.

freeze/cuda	addforce/cuda	addtorque	aveforce/cuda	enforce2d/cuda	gravity/cuda
npt/cuda	nve/cuda	nvt/cuda	setforce/cuda	shake/cuda	temp/berendsen/cuda
temp/rescale/cuda	temp/rescale/limit/cuda	viscous/cuda			

Compute styles

See the compute command for one−line descriptions of each style or click on the style itself for a full description:

angle/local	atom/molecule	bond/local	centro/atom	cluster/atom	cna/atom
com	com/molecule	coord/atom	damage/atom	dihedral/local	displace/atom
erotate/asphere	erotate/sphere	event/displace	group/group	gyration	gyration/molecule
heat/flux	improper/local	ke	ke/atom	msd	msd/molecule
pair	pair/local	pe	pe/atom	pressure	property/atom
property/local	property/molecule	rdf	reduce	reduce/region	slice
stress/atom	temp	temp/asphere	temp/com	temp/deform	temp/partial
temp/profile	temp/ramp	temp/region	temp/sphere	ti	

These are compute styles contributed by users, which can be used if DSMC is built with the appropriate package.

ackland/atom	ke/eff	ke/atom/eff	meso_e/atom	meso_rho/atom	meso_t/atom
temp/eff	temp/deform/eff	temp/region/eff	temp/rotate		

These are accelerated compute styles, which can be used if DSMC is built with the appropriate accelerated package.

pe/cuda	pressure/cuda	temp/cuda	temp/partial/cuda

Pair_style potentials

See the pair_style command for an overview of pair potentials. Click on the style itself for a full description:

none	hybrid	hybrid/overlay	adp
airebo	born	born/coul/long	buck
buck/coul/cut	buck/coul/long	colloid	comb
coul/cut	coul/debye	coul/long	dipole/cut
dpd	dpd/tstat	dsmc	eam
eam/alloy	eam/fs	eim	gauss
gayberne	gran/hertz/history	gran/hooke	gran/hooke/history
hbond/dreiding/lj	hbond/dreiding/morse	lj/charmm/coul/charmm	lj/charmm/coul/charmm/implicit
lj/charmm/coul/long	lj/class2	lj/class2/coul/cut	lj/class2/coul/long
lj/cut	lj/cut/coul/cut	lj/cut/coul/debye	lj/cut/coul/long
lj/cut/coul/long/tip4p	lj/expand	lj/gromacs	lj/gromacs/coul/gromacs
lj/smooth	lj96/cut	lubricate	meam
morse	peri/lps	peri/pmb	reax
rebo	resquared	soft	sw
table	tersoff	tersoff/zbl	yukawa
yukawa/colloid			

These are pair styles contributed by users, which can be used if DSMC is built with the appropriate package.

awpmd/cut	buck/coul	cg/cmm	cg/cmm/coul/cut
cg/cmm/coul/long	dipole/sf	eam/cd	edip

eff/cut	lj/coul	lj/sf	reax/csph/heatconduction
sph/idealgas	sph/lj	sph/rhosum	sph/taitwater
sph/taitwater/morris			

These are accelerated pair styles, which can be used if DSMC is built with the appropriate accelerated package.

born/coul/long/cuda	buck/coul/cut/cuda	buck/coul/long/cuda	buck/cuda
cg/cmm/coul/cut/cuda	cg/cmm/coul/debye/cuda	cg/cmm/coul/long/cuda	cg/cmm/coul/long/gpu
cg/cmm/cuda	cg/cmm/gpu	coul/long/gpu	eam/alloy/cuda
eam/alloy/opt	eam/cuda	eam/fs/cuda	eam/fs/opt
eam/opt	gayberne/gpu	gran/hooke/cuda	lj/charmm/coul/charmm/cuda
lj/charmm/coul/charmm/implicit/cuda	lj/charmm/coul/long/cuda	lj/charmm/coul/long/gpu	lj/charmm/coul/long/opt
lj/class2/coul/cut/cuda	lj/class2/coul/long/cuda	lj/class2/coul/long/gpu	lj/class2/cuda
lj/class2/gpu	lj/cut/coul/cut/cuda	lj/cut/coul/cut/gpu	lj/cut/coul/debye/cuda
lj/cut/coul/long/cuda	lj/cut/coul/long/gpu	lj/cut/cuda	lj/cut/experimental/cuda
lj/cut/gpu	lj/cut/opt	lj/expand/cuda	lj/expand/gpu
lj/gromacs/coul/gromacs/cuda	lj/gromacs/cuda	lj/smooth/cuda	lj96/cut/cuda
lj96/cut/gpu	morse/cuda	morse/gpu	morse/opt
resquared/gpu			

Bond_style potentials

See the bond_style command for an overview of bond potentials. Click on the style itself for a full description:

none	hybrid	class2	fene
fene/expand	harmonic	morse	nonlinear
quartic	table		

These are bond styles contributed by users, which can be used if DSMC is built with the appropriate package.

harmonic/shift	harmonic/shift/cut

Angle_style potentials

See the angle_style command for an overview of angle potentials. Click on the style itself for a full description:

none	hybrid	charmm	class2
cosine	cosine/delta	cosine/periodic	cosine/squared
harmonic	table		

These are angle styles contributed by users, which can be used if DSMC is built with the appropriate package.

cg/cmm	cosine/shift	cosine/shift/exp

Dihedral_style potentials

See the dihedral_style command for an overview of dihedral potentials. Click on the style itself for a full description:

none	hybrid	charmm	class2
harmonic	helix	multi/harmonic	opls

These are dihedral styles contributed by users, which can be used if DSMC is built with the appropriate package.

cosine/shift/exp

Improper_style potentials

See the improper_style command for an overview of improper potentials. Click on the style itself for a full description:

none	hybrid	class2	cvff
harmonic	umbrella		

Kspace solvers

See the kspace_style command for an overview of Kspace solvers. Click on the style itself for a full description:

ewald	pppm	pppm/cg	pppm/tip4p

These are Kspace solvers contributed by users, which can be used if DSMC is built with the appropriate package.

ewald/n

These are accelerated Kspace solvers, which can be used if DSMC is built with the appropriate accelerated package.

pppm/cuda	pppm/gpu

6. How−to discussions

The following sections describe how to use various options within DSMC.

The example input scripts included in the DSMC distribution and highlighted in this section also show how to setup and run various kinds of simulations.

6.1 Restarting a simulation

There are 3 ways to continue a long DSMC simulation. Multiple run commands can be used in the same input script. Each run will continue from where the previous run left off. Or binary restart files can be saved to disk using the restart command. At a later time, these binary files can be read via a read_restart command in a new script. Or they can be converted to text data files and read by a read_data command in a new script. This section discusses the *restart2data* tool that is used to perform the conversion.

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

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

```
restart         50 tmp.restart
```

added to it, it would produce 2 binary restart files (tmp.restart.50 and tmp.restart.100) as it ran.

This script could be used to read the 1st restart file and re−run the last 50 timesteps:

```
read_restart    tmp.restart.50

neighbor        0.4 bin
neigh_modify    every 1 delay 1

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

timestep        0.012

run             50
```

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

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

As an alternate approach, the restart file could be converted to a data file using this tool:

```
restart2data tmp.restart.50 tmp.restart.data
```

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

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

read_data       tmp.restart.data

neighbor        0.4 bin
neigh_modify    every 1 delay 1

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

timestep        0.012

reset_timestep  50
run             50
```

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

6.2 2d simulations

Use the dimension command to specify a 2d simulation.

Make the simulation box periodic in z via the boundary command. This is the default.

If using the create box command to define a simulation box, set the z dimensions narrow, but finite, so that the create_atoms command will tile the 3d simulation box with a single z plane of atoms − e.g.

```
create box 1 −10 10 −10 10 −0.25 0.25
```

If using the read data command to read in a file of atom coordinates, set the "zlo zhi" values to be finite but narrow, similar to the create_box command settings just described. For each atom in the file, assign a z coordinate so it falls inside the z−boundaries of the box − e.g. 0.0.

Use the fix enforce2d command as the last defined fix to insure that the z−components of velocities and forces are zeroed out every timestep. The reason to make it the last fix is so that any forces induced by other fixes will be zeroed out.

Many of the example input scripts included in the DSMC distribution are for 2d models.

IMPORTANT NOTE: Some models in DSMC treat particles as extended spheres, as opposed to point particles. In 2d, the particles will still be spheres, not disks, meaning their moment of inertia will be the same as in 3d.

6.3 CHARMM, AMBER, and DREIDING force fields

A force field has 2 parts: the formulas that define it and the coefficients used for a particular system. Here we only discuss formulas implemented in DSMC that correspond to formulas commonly used in the CHARMM, AMBER, and DREIDING force fields. Setting coefficients is done in the input data file via the read_data command or in the input script with commands like pair_coeff or bond_coeff. See this section for additional tools that can use CHARMM or AMBER to assign force field coefficients and convert their output into DSMC input.

See (MacKerell) for a description of the CHARMM force field. See (Cornell) for a description of the AMBER force field.

These style choices compute force field formulas that are consistent with common options in CHARMM or AMBER. See each command's documentation for the formula it computes.

- bond_style harmonic
- angle_style charmm
- dihedral_style charmm
- pair_style lj/charmm/coul/charmm
- pair_style lj/charmm/coul/charmm/implicit
- pair_style lj/charmm/coul/long

- special_bonds charmm
- special_bonds amber

DREIDING is a generic force field developed by the Goddard group at Caltech and is useful for predicting structures and dynamics of organic, biological and main−group inorganic molecules. The philosophy in DREIDING is to use general force constants and geometry parameters based on simple hybridization considerations, rather than individual force constants and geometric parameters that depend on the particular combinations of atoms involved in the bond, angle, or torsion terms. DREIDING has an explicit hydrogen bond term to describe interactions involving a hydrogen atom on very electronegative atoms (N, O, F).

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

These style choices compute force field formulas that are consistent with the DREIDING force field. See each command's documentation for the formula it computes.

- bond_style harmonic
- bond_style morse

- angle_style harmonic
- angle_style cosine
- angle_style cosine/periodic

- dihedral_style charmm
- improper_style umbrella

- pair_style buck
- pair_style buck/coul/cut
- pair_style buck/coul/long
- pair_style lj/cut
- pair_style lj/cut/coul/cut
- pair_style lj/cut/coul/long

- pair_style hbond/dreiding/lj
- pair_style hbond/dreiding/morse

- special_bonds dreiding

6.4 Running multiple simulations from one input script

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

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

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

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

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

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

```
read_data data.lj.new
run 10000
```

would run 2 independent simulations, one after the other.

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

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

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

```
variable a loop 8
variable t index 0.8 0.85 0.9 0.95 1.0 1.05 1.1 1.15
log log.$a
read data.polymer
velocity all create $t 352839
fix 1 all nvt $t $t 100.0
dump 1 all atom 1000 dump.$a
run 100000
next t
next a
jump in.polymer
```

All of the above examples work whether you are running on 1 or multiple processors, but assumed you are running DSMC on a single partition of processors. DSMC can be run on multiple partitions via the "−partition" command−line switch as described in this section of the manual.

In the last 2 examples, if DSMC were run on 3 partitions, the same scripts could be used if the "index" and "loop" variables were replaced with *universe*−style variables, as described in the variable command. Also, the "next t" and "next a" commands would need to be replaced with a single "next a t" command. With these modifications, the 8 simulations of each script would run on the 3 partitions one after the other until all were finished. Initially, 3 simulations would be started simultaneously, one on each partition. When one finished, that partition would then start the 4th simulation, and so forth, until all 8 were completed.

6.5 Multi−replica simulations

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

These are the relevant commands:

- neb for nudged elastic band calculations
- prd for parallel replica dynamics
- tad for temperature accelerated dynamics
- temper for parallel tempering

NEB is a method for finding transition states and barrier energies. PRD and TAD are methods for performing accelerated dynamics to find and perform infrequent events. Parallel tempering or replica exchange runs different replicas at a series of temperature to facilitate rare−event sampling.

These command can only be used if DSMC was built with the "replica" package. See the Making DSMC section for more info on packages.

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

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

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

Also note that with MPI installed on a machine (e.g. your desktop), you can run on more (virtual) processors than you have physical processors. Thus the above commands could be run on a single−processor (or few−processor) desktop so that you can run a multi−replica simulation on more replicas than you have physical processors.

6.6 Granular models

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

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

- atom_style sphere
- fix nve/sphere
- fix gravity

This compute

- compute erotate/sphere

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

Use one of these 3 pair potentials, which compute forces and torques between interacting pairs of particles:

- pair_style gran/history
- pair_style gran/no_history
- pair_style gran/hertzian

These commands implement fix options specific to granular systems:

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

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

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

- neigh_modify exclude

6.7 TIP3P water model

The TIP3P water model as implemented in CHARMM (MacKerell) specifies a 3−site rigid water molecule with charges and Lennard−Jones parameters assigned to each of the 3 atoms. In DSMC the fix shake command can be used to hold the two O−H bonds and the H−O−H angle rigid. A bond style of *harmonic* and an angle style of *harmonic* or *charmm* should also be used.

These are the additional parameters (in real units) to set for O and H atoms and the water molecule to run a rigid TIP3P−CHARMM model with a cutoff. The K values can be used if a flexible TIP3P model (without fix shake) is desired. If the LJ epsilon and sigma for HH and OH are set to 0.0, it corresponds to the original 1983 TIP3P model (Jorgensen).

O mass = 15.9994
H mass = 1.008

O charge = −0.834
H charge = 0.417

LJ epsilon of OO = 0.1521
LJ sigma of OO = 3.1507
LJ epsilon of HH = 0.0460
LJ sigma of HH = 0.4000
LJ epsilon of OH = 0.0836
LJ sigma of OH = 1.7753

K of OH bond = 450
r0 of OH bond = 0.9572

K of HOH angle = 55
theta of HOH angle = 104.52

These are the parameters to use for TIP3P with a long−range Coulombic solver (Ewald or PPPM in DSMC), see (Price) for details:

O mass = 15.9994
H mass = 1.008

O charge = −0.830
H charge = 0.415

LJ epsilon of OO = 0.102
LJ sigma of OO = 3.188
LJ epsilon, sigma of OH, HH = 0.0

K of OH bond = 450
r0 of OH bond = 0.9572

K of HOH angle = 55
theta of HOH angle = 104.52

Wikipedia also has a nice article on water models.

6.8 TIP4P water model

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

Currently, only a four−point model for long−range Coulombics is implemented via the DSMC pair style lj/cut/coul/long/tip4p. A cutoff version may be added the future. For both models, the bond lengths and bond angles should be held fixed using the fix shake command.

These are the additional parameters (in real units) to set for O and H atoms and the water molecule to run a rigid TIP4P model with a cutoff (Jorgensen). Note that the OM distance is specified in the pair_style command, not as part of the pair coefficients.

O mass = 15.9994
H mass = 1.008

O charge = −1.040
H charge = 0.520

r0 of OH bond = 0.9572
theta of HOH angle = 104.52

OM distance = 0.15

LJ epsilon of O−O = 0.1550
LJ sigma of O−O = 3.1536
LJ epsilon, sigma of OH, HH = 0.0

These are the parameters to use for TIP4P with a long−range Coulombic solver (Ewald or PPPM in DSMC):

O mass = 15.9994
H mass = 1.008

O charge = −1.0484
H charge = 0.5242

r0 of OH bond = 0.9572
theta of HOH angle = 104.52

OM distance = 0.1250

LJ epsilon of O−O = 0.16275
LJ sigma of O−O = 3.16435
LJ epsilon, sigma of OH, HH = 0.0

Wikipedia also has a nice article on water models.

6.9 SPC water model

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

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

O mass = 15.9994
H mass = 1.008

O charge = −0.820
H charge = 0.410

LJ epsilon of OO = 0.1553
LJ sigma of OO = 3.166
LJ epsilon, sigma of OH, HH = 0.0

r0 of OH bond = 1.0
theta of HOH angle = 109.47

Note that as originally proposed, the SPC model was run with a 9 Angstrom cutoff for both LJ and Coulommbic terms. It can also be used with long−range Coulombics (Ewald or PPPM in DSMC), without changing any of the parameters above, though it becomes a different model in that mode of usage.

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

O charge = −0.8476
H charge = 0.4238

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

Wikipedia also has a nice article on water models.

6.10 Coupling DSMC to other codes

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

DSMC can be coupled to other codes in at least 3 ways. Each has advantages and disadvantages, which you'll have to think about in the context of your application.

(1) Define a new fix command that calls the other code. In this scenario, DSMC is the driver code. During its timestepping, the fix is invoked, and can make library calls to the other code, which has been linked to DSMC as a library. This is the way the POEMS package that performs constrained rigid−body motion on groups of atoms is hooked to DSMC. See the fix_poems command for more details. See this section of the documentation for info on how to add a new fix to DSMC.

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

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

See this section of the documentation for how to add a new command to DSMC.

(3) Use DSMC as a library called by another code. In this case the other code is the driver and calls DSMC as needed. Or a wrapper code could link and call both DSMC and another code as libraries. Again, the run command has options that allow it to be invoked with minimal overhead (no setup or clean−up) if you wish to do multiple short runs, driven by another program.

DSMCes of driver codes that call DSMC as a library are included in the "couple" directory of the DSMC distribution; see couple/README for more details:

- simple: simple driver programs in C++ and C which invoke DSMC as a library
- lammps_quest: coupling of DSMC and Quest, to run classical MD with quantum forces calculated by a density functional code
- lammps_spparks: coupling of DSMC and SPPARKS, to couple a kinetic Monte Carlo model for grain growth using MD to calculate strain induced across grain boundaries

This section of the documentation describes how to build DSMC as a library. Once this is done, you can interface with DSMC either via C++, C, Fortran, or Python (or any other language that supports a vanilla C−like interface). For example, from C++ you could create one (or more) "instances" of DSMC, pass it an input script to process, or execute individual commands, all by invoking the correct class methods in DSMC. From C or Fortran you can make function calls to do the same things. See this section of the manual for a description of the Python wrapper provided with DSMC that operates through the DSMC library interface.

The files src/library.cpp and library.h contain the C−style interface to DSMC. See this section of the manual for a description of the interface and how to extend it for your needs.

Note that the lammps_open() function that creates an instance of DSMC takes an MPI communicator as an argument. This means that instance of DSMC will run on the set of processors in the communicator. Thus the calling code can run DSMC on all or a subset of processors. For example, a wrapper script might decide to alternate between DSMC and another code, allowing them both to run on all the processors. Or it might allocate half the processors to DSMC and half to the other code and run both codes simultaneously before syncing them up periodically. Or it might instantiate multiple instances of DSMC to perform different calculations.

6.11 Visualizing DSMC snapshots

DSMC itself does not do visualization, but snapshots from DSMC simulations can be visualized (and analyzed) in a variety of ways.

DSMC snapshots are created by the dump command which can create files in several formats. The native DSMC dump format is a text file (see "dump atom" or "dump custom") which can be visualized by the xmovie program, included with the DSMC package. This produces simple, fast 2d projections of 3d systems, and can be useful for rapid debugging of simulation geometry and atom trajectories.

Several programs included with DSMC as auxiliary tools can convert native DSMC dump files to other formats. See the Section_tools doc page for details. The first is the ch2lmp tool, which contains a lammps2pdb Perl script which converts DSMC dump files into PDB files. The second is the lmp2arc tool which converts DSMC dump files into Accelrys' Insight MD program files. The third is the lmp2cfg tool which converts DSMC dump files into CFG files which can be read into the AtomEye visualizer.

A Python−based toolkit distributed by our group can read native DSMC dump files, including custom dump files with additional columns of user−specified atom information, and convert them to various formats or pipe them into visualization software directly. See the Pizza.py WWW site for details. Specifically, Pizza.py can convert DSMC dump files into PDB, XYZ, Ensight, and VTK formats. Pizza.py can pipe DSMC dump files directly into the Raster3d and RasMol visualization programs. Pizza.py has tools that do interactive 3d OpenGL visualization and one that creates SVG images of dump file snapshots.

DSMC can create XYZ files directly (via "dump xyz") which is a simple text−based file format used by many visualization programs including VMD.

DSMC can create DCD files directly (via "dump dcd") which can be read by VMD in conjunction with a CHARMM PSF file. Using this form of output avoids the need to convert DSMC snapshots to PDB files. See the dump command for more information on DCD files.

DSMC can create XTC files directly (via "dump xtc") which is GROMACS file format which can also be read by VMD for visualization. See the dump command for more information on XTC files.

6.12 Triclinic (non−orthogonal) simulation boxes

By default, DSMC uses an orthogonal simulation box to encompass the particles. The boundary command sets the boundary conditions of the box (periodic, non−periodic, etc). The orthogonal box 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** = (0,yhi−ylo,0); **c** = (0,0,zhi−zlo). The 6 parameters (xlo,xhi,ylo,yhi,zlo,zhi) are defined at the time the simluation box is created, e.g. by the create_box or read_data or read_restart commands. Additionally, DSMC defines box size parameters lx,ly,lz where lx = xhi−xlo, and similarly in the y and z dimensions. The 6 parameters, as well as lx,ly,lz, can be output via the thermo_style custom command.

DSMC also allows simulations to be perfored in non−orthogonal simulation boxes shaped as a parallelepiped with triclinic symmetry. 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. Note that in DSMC the triclinic simulation box edge vectors **a**, **b**, and **c** cannot be arbitrary vectors. As indicated, **a** must be aligned with the x axis, **b** must be in the xy plane, and **c** is arbitrary. However, this is not a restriction since it is possible to rotate any set of 3 crystal basis vectors so that they meet this restriction.

The 9 parameters (xlo,xhi,ylo,yhi,zlo,zhi,xy,xz,yz) are defined at the time the simluation box is created. This happens in one of 3 ways. If the create_box command is used with a region of style *prism*, then a triclinic box is setup. See the region command for details. If the read_data command is used to define the simulation box, and the header of the data file contains a line with the "xy xz yz" keyword, then a triclinic box is setup. See the read_data command for details. Finally, if the read_restart command reads a restart file which was written from a simulation using a triclinic box, then a triclinic box will be setup for the restarted simulation.

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

As with orthogonal boxes, DSMC defines triclinic box size parameters lx,ly,lz where lx = xhi−xlo, and similarly in the y and z dimensions. The 9 parameters, as well as lx,ly,lz, can be output via the thermo_style custom command.

To avoid extremely tilted boxes (which would be computationally inefficient), no tilt factor can skew the box more than half the distance of the parallel box length, which is the 1st dimension in the tilt factor (x for xz). 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 geometrically all equivalent.

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

The inverse relationship can be written as follows:

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

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

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

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

```
xlo_bound = xlo + MIN(0.0,xy,xz,xy+xz)
xhi_bound = xhi + MAX(0.0,xy,xz,xy+xz)
ylo_bound = ylo + MIN(0.0,yz)
yhi_bound = yhi + MAX(0.0,yz)
zlo_bound = zlo
zhi_bound = zhi
```

These formulas can be inverted if you need to convert the bounding box back into the triclinic box parameters,

e.g. xlo = xlo_bound − MIN(0.0,xy,xz,xy+xz).

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

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

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

6.13 NEMD simulations

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

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

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

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

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

6.14 Extended spherical and aspherical particles

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

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

- atom styles
- pair potentials

- time integration
- computes, thermodynamics, and dump output
- rigid bodies composed of extended particles

Atom styles

There are 2 atom styles that allow for definition of finite−size particles: sphere and ellipsoid. The peri atom style also treats particles as having a volume, but that is internal to the pair_style peri potentials. The dipole atom style is most often used in conjunction with finite−size particles.

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

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

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

```
atom_style hybrid sphere dipole
```

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

Note that if one of these atom styles is used (or multiple styles via the atom_style hybrid command), not all particles in the system are required to be finite−size or aspherical. For example, if the 3 shape parameters are set to the same value, the particle will be a sphere rather than an ellipsoid. If the 3 shape parameters are all set to 0.0 or if the diameter is set to 0.0, it will be a point particle. If the length of the dipole moment is set to zero, the particle will not have a point dipole associated with it. The pair styles used to compute pairwise interactions will typically compute the correct interaction in these simplified (cheaper) cases. Pair_style hybrid can be used to insure the correct interactions are computed for the appropriate style of interactions. Likewise, using groups to partition particles (ellipsoids versus spheres versus point particles) will allow you to use the appropriate time integrators and temperature computations for each class of particles. See the doc pages for various commands for details.

Also note that for 2d simulations, finite−size spheres and ellipsoids are still treated as 3d particles, rather than as circular disks or ellipses. This means they have the same moment of inertia for a 3d extended object. When their temperature is coomputed, the correct degrees of freedom are used for rotation in a 2d versus 3d system.

Pair potentials

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

- pair_style gran/history
- pair_style gran/hertzian

- pair_style gran/no_history
- pair_style dipole/cut
- pair_style gayberne
- pair_style resquared
- pair_style lubricate

The granular pair styles are used with spherical particles. The dipole pair style is used with atom_style dipole, which could be applied to spherical or ellipsoidal particles. The GayBerne and REsquared potentials require ellipsoidal particles, though they will also work if the 3 shape parameters are the same (a sphere). The lubrication potential works with spherical particles.

Time integration

There are 3 fixes that perform time integration on extended spherical particles, meaning the integrators update the rotational orientation and angular velocity or angular momentum of the particles:

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

Likewise, there are 3 fixes that perform time integration on ellipsoids as extended aspherical particles:

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

The advantage of these fixes is that those which thermostat the particles include the rotational degrees of freedom in the temperature calculation and thermostatting. Other thermostats can be used with fix nve/sphere or fix nve/asphere, such as fix langevin or fix temp/berendsen, but those thermostats only operate on the translational kinetic energy of the extended particles.

Note that for mixtures of point and extended particles, you should only use these integration fixes on groups which contain extended particles.

Computes, thermodynamics, and dump output

There are 4 computes that calculate the temperature or rotational energy of extended spherical or aspherical particles (ellipsoids):

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

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

The dump custom command can output various attributes of extended particles, including the dipole moment (mu), the angular velocity (omega), the angular momentum (angmom), the quaternion (quat), and the torque (tq) on the particle.

Rigid bodies composed of extended particles

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

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

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

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

6.15 Output from DSMC (thermo, dumps, computes, fixes, variables)

There are four basic kinds of DSMC output:

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

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

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

The following sub−sections discuss different DSMC command related to output and the kind of data they operate on and produce:

- Global/per−atom/local data
- Scalar/vector/array data
- Thermodynamic output
- Dump file output
- Fixes that write output files
- Computes that process output quantities
- Fixes that process output quantities
- Computes that generate values to output
- Fixes that generate values to output
- Variables that generate values to output

- Summary table of output options and data flow between commands

Global/per−atom/local data

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

Scalar/vector/array data

Global, per−atom, and local datums can each come in three kinds: a single scalar value, a vector of values, or a 2d array of values. The doc page for a "compute" or "fix" or "variable" that generates data will specify both the style and kind of data it produces, e.g. a per−atom vector.

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

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 data once (vector −> scalar, array −> vector). Using two brackets reduces the dimension twice (array −> scalar). Thus a command that uses scalar values as input can typically also process elements of a vector or array.

Thermodynamic output

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

Dump file output

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

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

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

Fixes that write output files

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

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

The fix ave/spatial command enables direct output to a file of spatial−averaged per−atom quantities like those output in dump files, within 1d layers of the simulation box. The per−atom quantities can be atom density (mass or number) or atom attributes such as position, velocity, force. They can also be per−atom quantities calculated by a compute, by a fix, or by an atom−style variable. The spatial−averaged output of this fix can also be used as input to other output commands.

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

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

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

Computes that process output quantities

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

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

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

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

The compute atom/molecule command takes a list of one or more per−atom quantities (from a compute, fix, per−atom variable) and sums the quantities on a per−molecule basis. It produces a global vector or array as output

values which can be used as input to other output commands.

Fixes that process output quantities

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

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

Computes that generate values to output

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

Fixes that generate values to output

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

Variables that generate values to output

Every variables defined in an input script generates either a global scalar value or a per−atom vector (only atom−style variables) when it is accessed. The formulas used to define equal− and atom−style variables can contain references to the thermodynamic keywords and to global and per−atom data generated by computes, fixes, and other variables. The values generated by variables can be output using the other commands described in this section.

Summary table of output options and data flow between commands

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

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

Command	Input	Output
thermo_style custom	global scalars	screen, log file
dump custom	per−atom vectors	dump file
dump local	local vectors	dump file

fix print	global scalar from variable	screen, file
print	global scalar from variable	screen
computes	N/A	global/per−atom/local scalar/vector/array
fixes	N/A	global/per−atom/local scalar/vector/array
variables	global scalars, per−atom vectors	global scalar, per−atom vector
compute reduce	per−atom/local vectors	global scalar/vector
compute slice	global vectors/arrays	global vector/array
compute property/atom	per−atom vectors	per−atom vector/array
compute property/local	local vectors	local vector/array
compute atom/molecule	per−atom vectors	global vector/array
fix ave/atom	per−atom vectors	per−atom vector/array
fix ave/time	global scalars/vectors	global scalar/vector/array, file
fix ave/spatial	per−atom vectors	global array, file
fix ave/histo	global/per−atom/local scalars and vectors	global array, file
fix ave/correlate	global scalars	global array, file
fix store/state	per−atom vectors	per−atom vector/array

6.16 Thermostatting, barostatting, and computing temperature

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

Temperature is computed as kinetic energy divided by some number of degrees of freedom (and the Boltzmann constant). Since kinetic energy is a function of particle velocity, there is often a need to distinguish between a particle's advection velocity (due to some aggregate motiion of particles) and its thermal velocity. The sum of the two is the particle's total velocity, but the latter is often what is wanted to compute a temperature. DSMChas several options for computing temperatures, any of which can be used in thermostatting and barostatting. These compute commands calculate temperature, and the compute pressure command calculates pressure.

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

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

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

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

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

DPD thermostatting alters pairwise interactions in a manner analagous to the per−particle thermostatting of fix langevin.

Any of the thermostatting fixes can use temperature computes that remove bias for two purposes: (a) computing the current temperature to compare to the requested target temperature, and (b) adjusting only the thermal temperature component of the particle's velocities. See the doc pages for the individual fixes and for the fix_modify command for instructions on how to assign a temperature compute to a thermostatting fix. For example, you can apply a thermostat to only the x and z components of velocity by using it in conjunction with compute temp/partial.

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

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

Barostatting in DSMC is also performed by fixes. Two barosttating methods are currently available: Nose−Hoover (npt and nph) and Berendsen:

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

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

As with the thermostats, fix npt and fix nph only use translational motion of the particles in computing T and P and performing thermo/barostatting. Fix npt/sphere and fix npt/asphere thermo/barostat using not only translation

velocities but also rotational velocities for spherical and aspherical particles.

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

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

Finally, thermodynamic output, which can be setup via the thermo_style command, often includes temperature and pressure values. As explained on the doc page for the thermo_style command, the default T and P are setup by the thermo command itself. They are NOT the ones associated with any thermostatting or barostatting fix you have defined or with any compute that calculates a temperature or pressure. Thus if you want to view these values of T and P, you need to specify them explicitly via a thermo_style custom command. Or you can use the thermo_modify command to re−define what temperature or pressure compute is used for default thermodynamic output.

6.17 Walls

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

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

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

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

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

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

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

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

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

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

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

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

6.18 Elastic constants

Elastic constants characterize the stiffness of a material. The formal definition is provided by the linear relation that holds between the stress and strain tensors in the limit of infinitesimal deformation. In tensor notation, this is expressed as s_ij = C_ijkl * e_kl, where the repeated indices imply summation. s_ij are the elements of the symmetric stress tensor. e_kl are the elements of the symmetric strain tensor. C_ijkl are the elements of the fourth rank tensor of elastic constants. In three dimensions, this tensor has 3^4=81 elements. Using Voigt notation, the tensor can be written as a 6x6 matrix, where C_ij is now the derivative of s_i w.r.t. e_j. Because s_i is itself a derivative w.r.t. e_i, it follows that C_ij is also symmetric, with at most 7*6/2 = 21 distinct elements.

At zero temperature, it is easy to estimate these derivatives by deforming the cell in one of the six directions using the command displace_box and measuring the change in the stress tensor. A general−purpose script that does this is given in the examples/elastic directory described in this section.

Calculating elastic constants at finite temperature is more challenging, because it is necessary to run a simulation that perfoms time averages of differential properties. One way to do this is to measure the change in average stress tensor in an NVT simulations when the cell volume undergoes a finite deformation. In order to balance the systematic and statistical errors in this method, the magnitude of the deformation must be chosen judiciously, and care must be taken to fully equilibrate the deformed cell before sampling the stress tensor. Another approach is to sample the triclinic cell fluctuations that occur in an NPT simulation. This method can also be slow to converge and requires careful post−processing (Shinoda)

6.19 Library interface to DSMC

As described in this section, DSMC can be built as a library, so that it can be called by another code, used in a coupled manner with other codes, or driven through a Python interface.

All of these methodologies use a C−style interface to DSMC that is provided in the files src/library.cpp and src/library.h. The functions therein have a C−style argument list, but contain C++ code you could write yourself in a C++ application that was invoking DSMC directly. The C++ code in the functions illustrates how to invoke internal DSMC operations. Note that DSMC classes are defined within a DSMC namespace (DSMC_NS) if you use them from another C++ application.

Library.cpp contains these 4 functions:

```
void lammps_open(int, char **, MPI_Comm, void **);
void lammps_close(void *);
void lammps_file(void *, char *);
char *lammps_command(void *, char *);
```

The lammps_open() function is used to initialize DSMC, passing in a list of strings as if they were command−line arguments when DSMC is run in stand−alone mode from the command line, and a MPI communicator for DSMC to run under. It returns a ptr to the DSMC object that is created, and which is used in subsequent library calls. The lammps_open() function can be called multiple times, to create multiple instances of DSMC.

DSMC will run on the set of processors in the communicator. This means the calling code can run DSMC on all or a subset of processors. For example, a wrapper script might decide to alternate between DSMC and another code, allowing them both to run on all the processors. Or it might allocate half the processors to DSMC and half to the other code and run both codes simultaneously before syncing them up periodically. Or it might instantiate multiple instances of DSMC to perform different calculations.

The lammps_close() function is used to shut down an instance of DSMC and free all its memory.

The lammps_file() and lammps_command() functions are used to pass a file or string to DSMC as if it were an input script or single command in an input script. Thus the calling code can read or generate a series of DSMC commands one line at a time and pass it thru the library interface to setup a problem and then run it, interleaving the lammps_command() calls with other calls to extract information from DSMC, perform its own operations, or call another code's library.

Other useful functions are also included in library.cpp. For example:

```
void *lammps_extract_global(void *, char *)
void *lammps_extract_atom(void *, char *)
void *lammps_extract_compute(void *, char *, int, int)
void *lammps_extract_fix(void *, char *, int, int, int, int)
void *lammps_extract_variable(void *, char *, char *)
int lammps_get_natoms(void *)
void lammps_get_coords(void *, double *)
void lammps_put_coords(void *, double *)
```

These can extract various global or per−atom quantities from DSMC as well as values calculated by a compute, fix, or variable. The "get" and "put" operations can retrieve and reset atom coordinates. See the library.cpp file and its associated header file library.h for details.

The key idea of the library interface is that you can write any functions you wish to define how your code talks to DSMC and add them to src/library.cpp and src/library.h, as well as to the Python interface. The routines you add can access or change any DSMC data you wish. The couple and python directories have example C++ and C and Python codes which show how a driver code can link to DSMC as a library, run DSMC on a subset of processors, grab data from DSMC, change it, and put it back into DSMC.

6.20 Calculating thermal conductivity

The thermal conductivity kappa of a material can be measured in at least 3 ways using various options in DSMC. (See this section of the manual for an analogous discussion for viscosity). The thermal conducitivity tensor kappa is a measure of the propensity of a material to transmit heat energy in a diffusive manner as given by Fourier's law

J = −kappa grad(T)

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

The first method is to setup two thermostatted regions at opposite ends of a simulation box, or one in the middle and one at the end of a periodic box. By holding the two regions at different temperatures with a thermostatting fix, the energy added to the hot region should equal the energy subtracted from the cold region and be proportional to the heat flux moving between the regions. See the paper by Ikeshoji and Hafskjold for details of this idea. Note that thermostatting fixes such as fix nvt, fix langevin, and fix temp/rescale store the cumulative energy they add/subtract. Alternatively, the fix heat command can used in place of thermostats on each of two regions, and the resulting temperatures of the two regions monitored with the "compute temp/region" command or the temperature profile of the intermediate region monitored with the fix ave/spatial and compute ke/atom commands.

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

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

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

6.21 Calculating viscosity

The shear viscosity eta of a fluid can be measured in at least 3 ways using various options in DSMC. (See this section of the manual for an analogous discussion for thermal conductivity). Eta is a measure of the propensity of a fluid to transmit momentum in a direction perpendicular to the direction of velocity or momentum flow. Alternatively it is the resistance the fluid has to being sheared. It is given by

J = −eta grad(Vstream)

where J is the momentum flux in units of momentum per area per time. and grad(Vstream) is the spatial gradient of the velocity of the fluid moving in another direction, normal to the area through which the momentum flows. Viscosity thus has units of pressure−time.

The first method is to perform a non−equlibrium MD (NEMD) simulation by shearing the simulation box via the fix deform command, and using the fix nvt/sllod command to thermostat the fluid via the SLLOD equations of motion. The velocity profile setup in the fluid by this procedure can be monitored by the fix ave/spatial command, which determines grad(Vstream) in the equation above. E.g. the derivative in the y−direction of the Vx component of fluid motion or grad(Vstream) = dVx/dy. In this case, the Pxy off−diagonal component of the pressure or stress tensor, as calculated by the compute pressure command, can also be monitored, which is the J term in the equation above. See this section of the manual for details on NEMD simulations.

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

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

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

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

units       real
variable    T equal 86.4956
variable    V equal vol
variable    dt equal 4.0
variable    p equal 400     # correlation length
variable    s equal 5       # sample interval
variable    d equal $p*$s   # dump interval

# convert from DSMC real units to SI

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

# setup problem

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

# equilibration and thermalization

velocity     all create $T 102486 mom yes rot yes dist gaussian
fix          NVT all nvt temp $T $T 10 drag 0.2
run          8000
```

```
# viscosity calculation, switch to NVE if desired

#unfix       NVT
#fix         NVE all nve

reset_timestep 0
variable     pxy equal pxy
variable     pxz equal pxz
variable     pyz equal pyz
fix          SS all ave/correlate $s $p $d &             v_pxy v_pxz v_pyz type auto file S0St.dat a
variable     scale equal ${convert}/(${kB}*$T)*$V*$s*${dt}
variable     v11 equal trap(f_SS[3/)*${scale}
variable     v22 equal trap(f_SS[4/)*${scale}
variable     v33 equal trap(f_SS[5/)*${scale}
thermo_style custom step temp press v_pxy v_pxz v_pyz v_v11 v_v22 v_v33
run          100000
variable     v equal (v_v11+v_v22+v_v33)/3.0
variable     ndens equal count(all)/vol
print        "average viscosity: $v [Pa.s/ @ $T K, ${ndens} /A^3"
```

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

(Cornell) Cornell, Cieplak, Bayly, Gould, Merz, Ferguson, Spellmeyer, Fox, Caldwell, Kollman, JACS 117, 5179−5197 (1995).

(Horn) Horn, Swope, Pitera, Madura, Dick, Hura, and Head−Gordon, J Chem Phys, 120, 9665 (2004).

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

(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).

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

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

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

7. Example problems

The DSMC distribution includes an examples sub−directory with several sample problems. Each problem is in a sub−directory of its own. Most are 2d models so that they run quickly, requiring at most a couple of minutes to run on a desktop machine. Each problem has an input script (in.*) and produces a log file (log.*) and dump file (dump.*) when it runs. Some use a data file (data.*) of initial coordinates as additional input. A few sample log file outputs on different machines and different numbers of processors are included in the directories to compare your answers to. E.g. a log file like log.crack.foo.P means it ran on P processors of machine "foo".

The dump files produced by the example runs can be animated using the xmovie tool described in the Additional Tools section of the DSMC documentation. Animations of many of these examples can be viewed on the Movies section of the DSMC WWW Site.

These are the sample problems in the examples sub−directories:

colloid	big colloid particles in a small particle solvent, 2d system
comb	models using the COMB potential
crack	crack propagation in a 2d solid
dipole	point dipolar particles, 2d system
eim	NaCl using the EIM potential
ellipse	ellipsoidal particles in spherical solvent, 2d system
flow	Couette and Poiseuille flow in a 2d channel
friction	frictional contact of spherical asperities between 2d surfaces
indent	spherical indenter into a 2d solid
meam	MEAM test for SiC and shear (same as shear examples)
melt	rapid melt of 3d LJ system
micelle	self−assembly of small lipid−like molecules into 2d bilayers
min	energy minimization of 2d LJ melt
msst	MSST shock dynamics
neb	nudged elastic band (NEB) calculation for barrier finding
nemd	non−equilibrium MD of 2d sheared system
obstacle	flow around two voids in a 2d channel
peptide	dynamics of a small solvated peptide chain (5−mer)
peri	Peridynamic model of cylinder impacted by indenter
pour	pouring of granular particles into a 3d box, then chute flow
prd	parallel replica dynamics of a vacancy diffusion in bulk Si
reax	RDX and TATB models using the ReaxFF
rigid	rigid bodies modeled as independent or coupled

shear	sideways shear applied to 2d solid, with and without a void
srd	stochastic rotation dynamics (SRD) particles as solvent

Here is how you might run and visualize one of the sample problems:

```
cd indent
cp ../../src/lmp_linux .           # copy DSMC executable to this dir
lmp_linux <in.indent              # run the problem
```

Running the simulation produces the files *dump.indent* and *log.lammps*. You can visualize the dump file as follows:

```
../../tools/xmovie/xmovie −scale dump.indent
```

There is also an ELASTIC directory with an example script for computing elastic constants, using a zero temperature Si example. See the in.elastic file for more info.

There is also a USER directory which contains subdirectories of user−provided examples for user packages. See the README files in those directories for more info. See the doc/Section_start.html file for more info about user packages.

8. Performance &scalability

DSMC performance on several prototypical benchmarks and machines is discussed on the Benchmarks page of the DSMC WWW Site where CPU timings and parallel efficiencies are listed. Here, the benchmarks are described briefly and some useful rules of thumb about their performance are highlighted.

These are the 5 benchmark problems:

1. LJ = atomic fluid, Lennard−Jones potential with 2.5 sigma cutoff (55 neighbors per atom), NVE integration
2. Chain = bead−spring polymer melt of 100−mer chains, FENE bonds and LJ pairwise interactions with a 2^(1/6) sigma cutoff (5 neighbors per atom), NVE integration
3. EAM = metallic solid, Cu EAM potential with 4.95 Angstrom cutoff (45 neighbors per atom), NVE integration
4. Chute = granular chute flow, frictional history potential with 1.1 sigma cutoff (7 neighbors per atom), NVE integration
5. Rhodo = rhodopsin protein in solvated lipid bilayer, CHARMM force field with a 10 Angstrom LJ cutoff (440 neighbors per atom), particle−particle particle−mesh (PPPM) for long−range Coulombics, NPT integration

The input files for running the benchmarks are included in the DSMC distribution, as are sample output files. Each of the 5 problems has 32,000 atoms and runs for 100 timesteps. Each can be run as a serial benchmarks (on one processor) or in parallel. In parallel, each benchmark can be run as a fixed−size or scaled−size problem. For fixed−size benchmarking, the same 32K atom problem is run on various numbers of processors. For scaled−size benchmarking, the model size is increased with the number of processors. E.g. on 8 processors, a 256K−atom problem is run; on 1024 processors, a 32−million atom problem is run, etc.

A useful metric from the benchmarks is the CPU cost per atom per timestep. Since DSMC performance scales roughly linearly with problem size and timesteps, the run time of any problem using the same model (atom style, force field, cutoff, etc) can then be estimated. For example, on a 1.7 GHz Pentium desktop machine (Intel icc compiler under Red Hat Linux), the CPU run−time in seconds/atom/timestep for the 5 problems is

Problem:	LJ	Chain	EAM	Chute	Rhodopsin
CPU/atom/step:	4.55E−6	2.18E−6	9.38E−6	2.18E−6	1.11E−4
Ratio to LJ:	1.0	0.48	2.06	0.48	24.5

The ratios mean that if the atomic LJ system has a normalized cost of 1.0, the bead−spring chains and granular systems run 2x faster, while the EAM metal and solvated protein models run 2x and 25x slower respectively. The bulk of these cost differences is due to the expense of computing a particular pairwise force field for a given number of neighbors per atom.

Performance on a parallel machine can also be predicted from the one−processor timings if the parallel efficiency can be estimated. The communication bandwidth and latency of a particular parallel machine affects the efficiency. On most machines DSMC will give fixed−size parallel efficiencies on these benchmarks above 50% so long as the atoms/processor count is a few 100 or greater − i.e. on 64 to 128 processors. Likewise, scaled−size parallel efficiencies will typically be 80% or greater up to very large processor counts. The benchmark data on the DSMC WWW Site gives specific examples on some different machines, including a run of 3/4 of a billion LJ atoms on 1500 processors that ran at 85% parallel efficiency.

9. Additional tools

DSMC is designed to be a computational kernel for performing molecular dynamics computations. Additional pre− and post−processing steps are often necessary to setup and analyze a simulation. A few additional tools are provided with the DSMC distribution and are described in this section.

Our group has also written and released a separate toolkit called Pizza.py which provides tools for doing setup, analysis, plotting, and visualization for DSMC simulations. Pizza.py is written in Python and is available for download from the Pizza.py WWW site.

Note that many users write their own setup or analysis tools or use other existing codes and convert their output to a DSMC input format or vice versa. The tools listed here are included in the DSMC distribution as examples of auxiliary tools. Some of them are not actively supported by Sandia, as they were contributed by DSMC users. If you have problems using them, we can direct you to the authors.

The source code for each of these codes is in the tools sub−directory of the DSMC distribution. There is a Makefile (which you may need to edit for your platform) which will build several of the tools which reside in that directory. Some of them are larger packages in their own sub−directories with their own Makefiles.

- amber2lmp
- binary2txt
- ch2lmp
- chain
- createatoms
- data2xmovie
- eam database
- eam generate
- eff
- emacs
- ipp
- lmp2arc
- lmp2cfg
- lmp2vmd
- matlab
- micelle2d
- msi2lmp
- pymol_asphere
- python
- reax
- restart2data
- thermo_extract
- vim
- xmovie

amber2lmp tool

The amber2lmp sub−directory contains two Python scripts for converting files back−and−forth between the AMBER MD code and DSMC. See the README file in amber2lmp for more information.

These tools were written by Keir Novik while he was at Queen Mary University of London. Keir is no longer there and cannot support these tools which are out−of−date with respect to the current DSMC version (and maybe with respect to AMBER as well). Since we don't use these tools at Sandia, you'll need to experiment with them and make necessary modifications yourself.

---

binary2txt tool

The file binary2txt.cpp converts one or more binary DSMC dump file into ASCII text files. The syntax for running the tool is

```
binary2txt file1 file2 ...
```

which creates file1.txt, file2.txt, etc. This tool must be compiled on a platform that can read the binary file created by a DSMC run, since binary files are not compatible across all platforms.

---

ch2lmp tool

The ch2lmp sub−directory contains tools for converting files back−and−forth between the CHARMM MD code and DSMC.

They are intended to make it easy to use CHARMM as a builder and as a post−processor for DSMC. Using charmm2lammps.pl, you can convert an ensemble built in CHARMM into its DSMC equivalent. Using lammps2pdb.pl you can convert DSMC atom dumps into pdb files.

See the README file in the ch2lmp sub−directory for more information.

These tools were created by Pieter in't Veld (pjintve at sandia.gov) and Paul Crozier (pscrozi at sandia.gov) at Sandia.

---

chain tool

The file chain.f creates a DSMC data file containing bead−spring polymer chains and/or monomer solvent atoms. It uses a text file containing chain definition parameters as an input. The created chains and solvent atoms can strongly overlap, so DSMC needs to run the system initially with a "soft" pair potential to un−overlap it. The syntax for running the tool is

```
chain <def.chain > data.file
```

See the def.chain or def.chain.ab files in the tools directory for examples of definition files. This tool was used to create the system for the chain benchmark.

---

createatoms tool

The tools/createatoms directory contains a Fortran program called createAtoms.f which can generate a variety of interesting crystal structures and geometries and output the resulting list of atom coordinates in DSMC or other formats.

See the included Manual.pdf for details.

The tool is authored by Xiaowang Zhou (Sandia), xzhou at sandia.gov.

---

data2xmovie tool

The file data2xmovie.c converts a DSMC data file into a snapshot suitable for visualizing with the xmovie tool, as if it had been output with a dump command from DSMC itself. The syntax for running the tool is

```
data2xmovie options <infile > outfile
```

See the top of the data2xmovie.c file for a discussion of the options.

---

eam database tool

The tools/eam_database directory contains a Fortran program that will generate EAM alloy setfl potential files for any combination of 16 elements: Cu, Ag, Au, Ni, Pd, Pt, Al, Pb, Fe, Mo, Ta, W, Mg, Co, Ti, Zr. The files can then be used with the pair_style eam/alloy command.

The tool is authored by Xiaowang Zhou (Sandia), xzhou at sandia.gov, and is based on his paper:

X. W. Zhou, R. A. Johnson, and H. N. G. Wadley, Phys. Rev. B, 69, 144113 (2004).

---

eam generate tool

The tools/eam_generate directory contains several one−file C programs that convert an analytic formula into a tabulated embedded atom method (EAM) setfl potential file. The potentials they produce are in the potentials directory, and can be used with the pair_style eam/alloy command.

The source files and potentials were provided by Gerolf Ziegenhain (gerolf at ziegenhain.com).

---

eff tool

The tools/eff directory contains various scripts for generating structures and post−processing output for simulations using the electron force field (eFF).

These tools were provided by Andres Jaramillo−Botero at CalTech (ajaramil at wag.caltech.edu).

---

emacs tool

The tools/emacs directory contains a Lips add−on file for Emacs that enables a lammps−mode for editing of input scripts when using Emacs, with various highlighting options setup.

These tools were provided by Aidan Thompson at Sandia (athomps at sandia.gov).

---

ipp tool

The tools/ipp directory contains a Perl script ipp which can be used to facilitate the creation of a complicated file (say, a lammps input script or tools/createatoms input file) using a template file.

ipp was created and is maintained by Reese Jones (Sandia), rjones at sandia.gov.

See two examples in the tools/ipp directory. One of them is for the tools/createatoms tool's input file.

---

lmp2arc tool

The lmp2arc sub−directory contains a tool for converting DSMC output files to the format for Accelrys' Insight MD code (formerly MSI/Biosym and its Discover MD code). See the README file for more information.

This tool was written by John Carpenter (Cray), Michael Peachey (Cray), and Steve Lustig (Dupont). John is now at the Mayo Clinic (jec at mayo.edu), but still fields questions about the tool.

This tool was updated for the current DSMC C++ version by Jeff Greathouse at Sandia (jagreat at sandia.gov).

---

lmp2cfg tool

The lmp2cfg sub−directory contains a tool for converting DSMC output files into a series of *.cfg files which can be read into the AtomEye visualizer. See the README file for more information.

This tool was written by Ara Kooser at Sandia (askoose at sandia.gov).

---

lmp2vmd tool

The lmp2vmd sub−directory contains a README.txt file that describes details of scripts and plugin support within the VMD package for visualizing DSMC dump files.

The VMD plugins and other supporting scripts were written by Axel Kohlmeyer (akohlmey at cmm.chem.upenn.edu) at U Penn.

---

matlab tool

The matlab sub−directory contains several MATLAB scripts for post−processing DSMC output. The scripts include readers for log and dump files, a reader for EAM potential files, and a converter that reads DSMC dump files and produces CFG files that can be visualized with the AtomEye visualizer.

See the README.pdf file for more information.

These scripts were written by Arun Subramaniyan at Purdue Univ (asubrama at purdue.edu).

---

micelle2d tool

The file micelle2d.f creates a DSMC data file containing short lipid chains in a monomer solution. It uses a text file containing lipid definition parameters as an input. The created molecules and solvent atoms can strongly overlap, so DSMC needs to run the system initially with a "soft" pair potential to un−overlap it. The syntax for running the tool is

```
micelle2d <def.micelle2d > data.file
```

See the def.micelle2d file in the tools directory for an example of a definition file. This tool was used to create the system for the micelle example.

---

msi2lmp tool

The msi2lmp sub−directory contains a tool for creating DSMC input data files from Accelrys' Insight MD code (formerly MSI/Biosym and its Discover MD code). See the README file for more information.

This tool was written by John Carpenter (Cray), Michael Peachey (Cray), and Steve Lustig (Dupont). John is now at the Mayo Clinic (jec at mayo.edu), but still fields questions about the tool.

This tool may be out−of−date with respect to the current DSMC and Insight versions. Since we don't use it at Sandia, you'll need to experiment with it yourself.

---

pymol_asphere tool

The pymol_asphere sub−directory contains a tool for converting a DSMC dump file that contains orientation info for ellipsoidal particles into an input file for the PyMol visualization package.

Specifically, the tool triangulates the ellipsoids so they can be viewed as true ellipsoidal particles within PyMol. See the README and examples directory within pymol_asphere for more information.

This tool was written by Mike Brown at Sandia.

---

python tool

The python sub−directory contains several Python scripts that perform common DSMC post−processing tasks, such as:

- extract thermodynamic info from a log file as columns of numbers
- plot two columns of thermodynamic info from a log file using GnuPlot
- sort the snapshots in a dump file by atom ID
- convert multiple NEB dump files into one dump file for viz
- convert dump files into XYZ, CFG, or PDB format for viz by other packages

These are simple scripts built on Pizza.py modules. See the README for more info on Pizza.py and how to use these scripts.

---

reax tool

The reax sub−directory contains stand−alond codes that can post−process the output of the fix reax/bonds command from a DSMC simulation using ReaxFF. See the README.txt file for more info.

These tools were written by Aidan Thompson at Sandia.

---

restart2data tool

The file restart2data.cpp converts a binary DSMC restart file into an ASCII data file. The syntax for running the tool is

```
restart2data restart−file data−file (input−file)
```

Input−file is optional and if specified will contain DSMC input commands for the masses and force field parameters, instead of putting those in the data−file. Only a few force field styles currently support this option.

This tool must be compiled on a platform that can read the binary file created by a DSMC run, since binary files are not compatible across all platforms.

Note that a text data file has less precision than a binary restart file. Hence, continuing a run from a converted data file will typically not conform as closely to a previous run as will restarting from a binary restart file.

If a "%" appears in the specified restart−file, the tool expects a set of multiple files to exist. See the restart and write_restart commands for info on how such sets of files are written by DSMC, and how the files are named.

---

thermo_extract tool

The thermo_extract tool reads one of more DSMC log files and extracts a thermodynamic value (e.g. Temp, Press). It spits out the time,value as 2 columns of numbers so the tool can be used as a quick way to plot some quantity of interest. See the header of the thermo_extract.c file for the syntax of how to run it and other details.

This tool was written by Vikas Varshney at Wright Patterson AFB (vikas.varshney at gmail.com).

---

vim tool

The files in the tools/vim directory are add−ons to the VIM editor that allow easier editing of DSMC input scripts. See the README.txt file for details.

These files were provided by Gerolf Ziegenhain (gerolf at ziegenhain.com)

---

xmovie tool

The xmovie tool is an X−based visualization package that can read DSMC dump files and animate them. It is in its own sub−directory with the tools directory. You may need to modify its Makefile so that it can find the appropriate X libraries to link against.

The syntax for running xmovie is

```
xmovie options dump.file1 dump.file2 ...
```

If you just type "xmovie" you will see a list of options. Note that by default, DSMC dump files are in scaled coordinates, so you typically need to use the −scale option with xmovie. When xmovie runs it opens a visualization window and a control window. The control options are straightforward to use.

Xmovie was mostly written by Mike Uttormark (U Wisconsin) while he spent a summer at Sandia. It displays 2d projections of a 3d domain. While simple in design, it is an amazingly fast program that can render large numbers of atoms very quickly. It's a useful tool for debugging DSMC input and output and making sure your simulation is doing what you think it should. The animations on the Examples page of the DSMC WWW site were created with xmovie.

I've lost contact with Mike, so I hope he's comfortable with us distributing his great tool!

10. Modifying &extending DSMC

DSMC is designed in a modular fashion so as to be easy to modify and extend with new functionality. In fact, about 75% of its source code is files added in this fashion.

In this section, changes and additions users can make are listed along with minimal instructions. If you add a new feature to DSMC and think it will be of interest to general users, we encourage you to submit it to the developers for inclusion in the released version of DSMC. Information about how to do this is provided below.

The best way to add a new feature is to find a similar feature in DSMC and look at the corresponding source and header files to figure out what it does. You will need some knowledge of C++ to be able to understand the hi−level structure of DSMC and its class organization, but functions (class methods) that do actual computations are written in vanilla C−style code and operate on simple C−style data structures (vectors and arrays).

Most of the new features described in this section require you to write a new C++ derived class (except for exceptions described below, where you can make small edits to existing files). Creating a new class requires 2 files, a source code file (*.cpp) and a header file (*.h). The derived class must provide certain methods to work as a new option. Depending on how different your new feature is compared to existing features, you can either derive from the base class itself, or from a derived class that already exists. Enabling DSMC to invoke the new class is as simple as putting the two source files in the src dir and re−building DSMC.

The advantage of C++ and its object−orientation is that all the code and variables needed to define the new feature are in the 2 files you write, and thus shouldn't make the rest of DSMC more complex or cause side−effect bugs.

Here is a concrete example. Suppose you write 2 files pair_foo.cpp and pair_foo.h that define a new class PairFoo that computes pairwise potentials described in the classic 1997 paper by Foo, et al. If you wish to invoke those potentials in a DSMC input script with a command like

```
pair_style foo 0.1 3.5
```

then your pair_foo.h file should be structured as follows:

```
#ifdef PAIR_CLASS
PairStyle(foo,PairFoo)
#else
...
(class definition for PairFoo)
...
#endif
```

where "foo" is the style keyword in the pair_style command, and PairFoo is the class name defined in your pair_foo.cpp and pair_foo.h files.

When you re−build DSMC, your new pairwise potential becomes part of the executable and can be invoked with a pair_style command like the example above. Arguments like 0.1 and 3.5 can be defined and processed by your new class.

As illustrated by this pairwise example, many kinds of options are referred to in the DSMC documentation as the "style" of a particular command.

The instructions below give the header file for the base class that these styles are derived from. Public variables in that file are ones used and set by the derived classes which are also used by the base class. Sometimes they are also used by the rest of DSMC. Virtual functions in the base class header file which are set = 0 are ones you must define in your new derived class to give it the functionality DSMC expects. Virtual functions that are not set to 0 are functions you can optionally define.

Additionally, new output options can be added directly to the thermo.cpp, dump_custom.cpp, and variable.cpp files as explained below.

Here are additional guidelines for modifying DSMC and adding new functionality:

- Think about whether what you want to do would be better as a pre− or post−processing step. Many computations are more easily and more quickly done that way.
- Don't do anything within the timestepping of a run that isn't parallel. E.g. don't accumulate a bunch of data on a single processor and analyze it. You run the risk of seriously degrading the parallel efficiency.
- If your new feature reads arguments or writes output, make sure you follow the unit conventions discussed by the units command.
- If you add something you think is truly useful and doesn't impact DSMC performance when it isn't used, send an email to the developers. We might be interested in adding it to the DSMC distribution. See further details on this at the bottom of this page.

Here are the subsequent topics discussed below, most of which are new features that can be added in the manner just described:

10.1 Atom styles

Classes that define an atom style are derived from the Atom class. The atom style determines what quantities are associated with an atom. A new atom style can be created if one of the existing atom styles does not define all the arrays you need to store and communicate with atoms.

Atom_vec_atomic.cpp is a simple example of an atom style.

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

grow	re−allocate atom arrays to longer lengths
copy	copy info for one atom to another atom's array locations
pack_comm	store an atom's info in a buffer communicated every timestep
pack_comm_vel	add velocity info to buffer
pack_comm_one	store extra info unique to this atom style
unpack_comm	retrieve an atom's info from the buffer
unpack_comm_vel	also retrieve velocity info
unpack_comm_one	retreive extra info unique to this atom style
pack_reverse	store an atom's info in a buffer communicating partial forces
pack_reverse_one	store extra info unique to this atom style
unpack_reverse	retrieve an atom's info from the buffer
unpack_reverse_one	retreive extra info unique to this atom style
pack_border	store an atom's info in a buffer communicated on neighbor re−builds
pack_border_vel	add velocity info to buffer
pack_border_one	store extra info unique to this atom style
unpack_border	retrieve an atom's info from the buffer
unpack_border_vel	also retrieve velocity info
unpack_border_one	retreive extra info unique to this atom style
pack_exchange	store all an atom's info to migrate to another processor
unpack_exchange	retrieve an atom's info from the buffer
size_restart	number of restart quantities associated with proc's atoms
pack_restart	pack atom quantities into a buffer
unpack_restart	unpack atom quantities from a buffer
create_atom	create an individual atom of this style
data_atom	parse an atom line from the data file
memory_usage	tally memory allocated by atom arrays

The constructor of the derived class sets values for several variables that you must set when defining a new atom style, which are documented in atom_vec.h. New atom arrays are defined in atom.cpp. Search for the word "customize" and you will find locations you will need to modify.

10.2 Bond, angle, dihedral, improper potentials

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 DSMC.

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 methods you define in your new derived bond class. See bond.h, angle.h, dihedral.h, and improper.h for details.

compute	compute the molecular interactions
coeff	

	set coefficients for one bond type
equilibrium_distance	length of bond, used by SHAKE
write &read_restart	writes/reads coeffs to restart files
single	force and energy of a single bond

10.3 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 DSMC.

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.

compute_scalar	compute a scalar quantity
compute_vector	compute a vector of quantities
compute_peratom	compute one or more quantities per atom
pack_comm	pack a buffer with items to communicate
unpack_comm	unpack the buffer
pack_reverse	pack a buffer with items to reverse communicate
unpack_reverse	unpack the buffer
memory_usage	tally memory usage

10.4 Dump styles

10.5 Dump custom output options

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.

Alternatively, you can add new keywords to the dump custom command. Search for the word "customize" in dump_custom.cpp to see the half−dozen or so locations where code will need to be added.

10.6 Fix styles

In DSMC, 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 DSMC.

Fix_setforce.cpp is a simple example of setting forces on atoms to prescribed values. There are dozens of fix options already in DSMC; 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
init	initialization before a run
setup	called immediately before the 1st timestep
initial_integrate	called at very beginning of each timestep
pre_exchange	called before atom exchange on re−neighboring steps
pre_neighbor	called before neighbor list build
post_force	called after pair &molecular forces are computed
final_integrate	called at end of each timestep
end_of_step	called at very end of timestep
write_restart	dumps fix info to restart file
restart	uses info from restart file to re−initialize the fix
grow_arrays	allocate memory for atom−based arrays used by fix
copy_arrays	copy atom info when an atom migrates to a new processor
memory_usage	report memory used by fix
pack_exchange	store atom's data in a buffer
unpack_exchange	retrieve atom's data from a buffer
pack_restart	store atom's data for writing to restart file
unpack_restart	retrieve atom's data from a restart file buffer
size_restart	size of atom's data
maxsize_restart	max size of atom's data
initial_integrate_respa	same as initial_integrate, but for rRESPA
post_force_respa	same as post_force, but for rRESPA
final_integrate_respa	same as final_integrate, but for rRESPA
pack_comm	pack a buffer to communicate a per−atom quantity
unpack_comm	unpack a buffer to communicate a per−atom quantity

pack_reverse_comm	pack a buffer to reverse communicate a per−atom quantity
unpack_reverse_comm	unpack a buffer to reverse communicate a per−atom quantity
thermo	compute quantities for thermodynamic output

Typically, only a small fraction of these methods are defined for a particular fix. Setmask is mandatory, as it determines when the fix will be invoked during the timestep. Fixes that perform time integration (*nve*, *nvt*, *npt*) implement 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.

10.7 Input script commands

New commands can be added to DSMC input scripts by adding new classes that 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 DSMC input script, DSMC simply creates a class 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 DSMC 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.

10.8 Kspace computations

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 DSMC.

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 1st timestep of a run

compute	every−timestep computation
memory_usage	tally of memory usage

10.9 Minimization styles

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

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

10.10 Pairwise potentials

Classes that compute pairwise interactions are derived from the Pair class. In DSMC, pairwise calculation include manybody potentials such as EAM or Tersoff where particles interact without a static bond topology. New styles can be created to add new pair potentials to DSMC.

Pair_lj_cut.cpp is a simple example of a Pair class, though it includes some optional methods to enable its use with rRESPA.

Here is a brief description of the class methods in pair.h:

compute	workhorse routine that computes pairwise interactions
settings	reads the input script line with arguments you define
coeff	set coefficients for one i,j type pair
init_one	perform initialization for one i,j type pair
init_style	initialization specific to this pair style
write &read_restart	write/read i,j pair coeffs to restart files
write &read_restart_settings	write/read global settings to restart files
single	force and energy of a single pairwise interaction between 2 atoms
compute_inner/middle/outer	versions of compute used by rRESPA

The inner/middle/outer routines are optional.

10.11 Region styles

Classes that define geometric regions are derived from the Region class. Regions are used elsewhere in DSMC 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 DSMC.

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.

match	determine whether a point is in the region

10.12 Thermodynamic output options

There is one class that computes and prints thermodynamic information to the screen and log file; see the file thermo.cpp.

There are two styles defined in thermo.cpp: "one" and "multi". There is also a flexible "custom" style which allows the user to explicitly list keywords for quantities to print when thermodynamic info is output. See the thermo_style command for a list of defined quantities.

The thermo styles (one, multi, etc) are simply lists of keywords. Adding a new style thus only requires defining a new list of keywords. Search for the word "customize" with references to "thermo style" in thermo.cpp to see the two locations where code will need to be added.

New keywords can also be added to thermo.cpp to compute new quantities for output. Search for the word "customize" with references to "keyword" in thermo.cpp to see the several locations where code will need to be added.

Note that the thermo_style custom command already allows for thermo output of quantities calculated by fixes, computes, and variables. Thus, it may be simpler to compute what you wish via one of those constructs, than by adding a new keyword to the thermo command.

10.13 Variable options

There is one class that computes and stores variable information in DSMC; see the file variable.cpp. The value associated 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) was discussed here on this page.

Adding a new math function of one or two arguments can be done by editing one section of the Variable::evaulate() 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::evaulate() 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::evaulate() method. Search for the word "customize" to find the appropriate location.

Adding new compute styles (whose calculated values can then be accessed by variables) was discussed here on this page.

10.14 Submitting new features for inclusion in DSMC

We encourage users to submit new features that they add to DSMC to the developers, especially if you think the features will be of interest to other users. If they are broadly useful we may add them as core files to DSMC or as part of a standard package. Else we will add them as a user−contributed package or file. Examples of user packages are in src sub−directories that start with USER. The USER−MISC package is simply a collection of (mostly) unrelated single files, which is the simplest way to have your contribution quickly added to the DSMC distribution. You can see a list of the both standard and user packages by typing "make package" in the DSMC src directory.

With user packages and files, all we are really providing (aside from the fame and fortune that accompanies having your name in the source code and on the Authors page of the DSMC WWW site), is a means for you to distribute your work to the DSMC user community and a mechanism for others to easily try out your new feature. This may help you find bugs or make contact with new collaborators. Note that you're also implicitly agreeing to support your code which means answer questions, fix bugs, and maintain it if DSMC changes.

The previous sections of this doc page describe how to add new features of various kinds to DSMC. Packages are simply collections of one or more new class files which are invoked as a new "style" within a DSMC input script. If designed correctly, these additions do not require changes to the main core of DSMC; they are simply add−on files. If you think your new feature requires non−trivial changes in core DSMC files, you'll need to communicate with the developers, since we may or may not want to make those changes. An example of a trivial change is making a parent−class method "virtual" when you derive a new child class from it.

Here is what you need to do to submit a user package or single file for our consideration. Following these steps will save time for both you and us. See existing package files for examples.

- All source files you provide must compile with the most current version of DSMC.
- If your contribution is a single file (actually a *.cpp and *.h file) it can most rapidly be added to the USER−MISC directory. Send us the one−line entry to add to the USER−MISC/README file in that dir, along with the 2 source files. You can do this multiple times if you wish to contribute several individual features.
- If your contribution is several related featues, it is probably best to make it a user package directory with a name like USER−FOO. In addition to your new files, the directory should contain a README, and Install.csh file. The README text file should contain your name and contact information and a brief description of what your new package does. The Install.csh file enables DSMC to include and exclude your package. See other README and Install.sh files in other USER directories as examples. Send us a tarball of this USER−FOO directory.
- Your new source files need to have the DSMC copyright, GPL notice, and your name at the top, like other DSMC source files. They need to create a class that is inside the DSMC namespace. Other than that, your files can do whatever is necessary to implement the new features. They don't have to be written in the same stylistic format and syntax as other DSMC files, though that would be nice.
- Finally, you must also send a documentation file for each new command or style you are adding to DSMC. This will be one file for a single−file feature. For a package, it might be several files. These are simple text files which we will convert to HTML. They must be in the same format as other *.txt files in the lammps/doc directory for similar commands and styles. The "Restrictions" section of the doc page should indicate that your command is only available if DSMC is built with the appropriate USER−MISC or USER−FOO package. See other user package doc files for an example of how to do this. The txt2html

tool we use to do the conversion can be downloaded from this site, so you can perform the HTML conversion yourself to proofread your doc page.

Note that the more clear and self−explanatory you make your doc and README files, the more likely it is that users will try out your new feature.

(Foo) Foo, Morefoo, and Maxfoo, J of Classic Potentials, 75, 345 (1997).

11. Python interface to DSMC

The DSMC distribution includes some Python code in its python directory which wraps the library interface to DSMC. This makes it is possible to run DSMC, invoke DSMC commands or give it an input script, extract DSMC results, an modify internal DSMC variables, either from a Python script or interactively from a Python prompt.

Python is a powerful scripting and programming language which can be used to wrap software like DSMC and other packages. It can be used to glue multiple pieces of software together, e.g. to run a coupled or multiscale model. See this section of the manual and the couple directory of the distribution for more ideas about coupling DSMC to other codes. See this section about how to build DSMC as a library, and this section for a description of the library interface provided in src/library.cpp and src/library.h and how to extend it for your needs. As described below, that interface is what is exposed to Python. It is designed to be easy to add functions to. This has the effect of extending the Python inteface as well. See details below.

By using the Python interface DSMC can also be coupled with a GUI or visualization tools that display graphs or animations in real time as DSMC runs. Examples of such scripts are inlcluded in the python directory.

Two advantages of using Python are how concise the language is and that it can be run interactively, enabling rapid development and debugging of programs. If you use it to mostly invoke costly operations within DSMC, such as running a simulation for a reasonable number of timesteps, then the overhead cost of invoking DSMC thru Python will be negligible.

Before using DSMC from a Python script, the Python on your machine must be "extended" to include an interface to the DSMC library. If your Python script will invoke MPI operations, you will also need to extend your Python with an interface to MPI itself.

Thus you should first decide how you intend to use DSMC from Python. There are 3 options:

(1) Use DSMC on a single processor running Python.

(2) Use DSMC in parallel, where each processor runs Python, but your Python program does not use MPI.

(3) Use DSMC in parallel, where each processor runs Python, and your Python script also makes MPI calls through a Python/MPI interface.

Note that for (2) and (3) you will not be able to use Python interactively by typing commands and getting a response. This is because you will have multiple instances of Python running (e.g. on a parallel machine) and they cannot all read what you type.

Working in mode (1) does not require your machine to have MPI installed. You should extend your Python with a serial version of DSMC and the dummy MPI library provided with DSMC. See instructions below on how to do this.

Working in mode (2) requires your machine to have an MPI library installed, but your Python does not need to be extended with MPI itself. The MPI library must be a shared library (e.g. a *.so file on Linux) which is not typically created when MPI is built/installed. See instruction below on how to do this. You should extend your Python with the a parallel versionn of DSMC which will use the shared MPI system library. See instructions below on how to do this.

Working in mode (3) requires your machine to have MPI installed (as a shared library as in (2)). You must also extend your Python with a parallel version of DSMC (same as in (2)) and with MPI itself, via one of several available Python/MPI packages. See instructions below on how to do the latter task.

Several of the following sub−sections cover the rest of the Python setup discussion. The next to last sub−section describes the Python syntax used to invoke DSMC. The last sub−section describes example Python scripts included in the python directory.

Before proceeding, there are 2 items to note.

(1) The provided Python wrapper for DSMC uses the amazing and magical (to me) "ctypes" package in Python, which auto−generates the interface code needed between Python and a set of C interface routines for a library. Ctypes is part of standard Python for versions 2.5 and later. You can check which version of Python you have installed, by simply typing "python" at a shell prompt.

(2) Any library wrapped by Python, including DSMC, must be built as a shared library (e.g. a *.so file on Linux and not a *.a file). The python/setup_serial.py and setup.py scripts do this build for DSMC itself (described below). But if you have DSMC configured to use additional packages that have their own libraries, then those libraries must also be shared libraries. E.g. MPI, FFTW, or any of the libraries in lammps/lib. When you build DSMC as a stand−alone code, you are not building shared versions of these libraries.

The discussion below describes how to create a shared MPI library. I suggest you start by configuing DSMC without packages installed that require any libraries besides MPI. See this section of the manual for a discussion of DSMC pacakges. E.g. do not use the KSPACE, GPU, MEAM, POEMS, or REAX packages.

If you are successfully follow the steps belwo to build the Python wrappers and use this version of DSMC through Python, you can then take the next step of adding DSMC packages that use additional libraries. This will require you to build a shared library for that package's library, similar to what is described below for MPI. It will also require you to edit the python/setup_serial.py or setup.py scripts to enable Python to access those libraries when it builds the DSMC wrapper.

11.1 Extending Python with a serial version of DSMC

From the python directory in the DSMC distribution, type

```
python setup_serial.py build
```

and then one of these commands:

```
sudo python setup_serial.py install
python setup_serial.py install −−home=~/foo
```

The "build" command should compile all the needed DSMC files, including its dummy MPI library. The first "install" command will put the needed files in your Python's site−packages sub−directory, so that Python can load them. For example, if you installed Python yourself on a Linux machine, it would typically be somewhere like /usr/local/lib/python2.5/site−packages. Installing Python packages this way often requires you to be able to write to the Python directories, which may require root priveleges, hence the "sudo" prefix. If this is not the case, you can drop the "sudo".

Alternatively, you can install the DSMC files (or any other Python packages) in your own user space. The second "install" command does this, where you should replace "foo" with your directory of choice.

If these commands are successful, a *lammps.py* and *_lammps_serial.so* file will be put in the appropriate directory.

11.2 Creating a shared MPI library

A shared library is one that is dynamically loadable, which is what Python requires. On Linux this is a library file that ends in ".so", not ".a". Such a shared library is normally not built if you installed MPI yourself, but it is easy to do. Here is how to do it for MPICH, a popular open−source version of MPI, distributed by Argonne National Labs. From within the mpich directory, type

```
./configure −−enable−sharedlib=gcc
make
make install
```

You may need to use "sudo make install" in place of the last line. The end result should be the file libmpich.so in /usr/local/lib.

IMPORTANT NOTE: If the file libmpich.a already exists in your installation directory (e.g. /usr/local/lib), you will now have both a static and shared MPI library. This will be fine for running DSMC from Python since it only uses the shared library. But if you now try to build DSMC by itself as a stand−alone program (cd lammps/src; make foo) or build other codes that expect to link against libmpich.a, then those builds will typically fail if the linker uses libmpich.so instead. This means you will need to remove the file /usr/local/lib/libmich.so before building DSMC again as a stand−alone code.

11.3 Extending Python with a parallel version of DSMC

From the python directory, type

```
python setup.py build
```

and then one of these commands:

```
sudo python setup.py install
python setup.py install −−home=~/foo
```

The "build" command should compile all the needed DSMC C++ files, which will require MPI to be installed on your system. This means it must find both the header file mpi.h and a shared library file, e.g. libmpich.so if the MPICH version of MPI is installed. See the preceding section for how to create a shared library version of MPI if it does not exist. You may need to adjust the "include_dirs" and "library_dirs" and "libraries" fields in python/setup.py to insure the Python build finds all the files it needs.

The first "install" command will put the needed files in your Python's site−packages sub−directory, so that Python can load them. For example, if you installed Python yourself on a Linux machine, it would typically be somewhere like /usr/local/lib/python2.5/site−packages. Installing Python packages this way often requires you to be able to write to the Python directories, which may require root priveleges, hence the "sudo" prefix. If this is not the case, you can drop the "sudo".

Alternatively, you can install the DSMC files (or any other Python packages) in your own user space. The second "install" command does this, where you should replace "foo" with your directory of choice.

If these commands are successful, a *lammps.py* and *_lammps.so* file will be put in the appropriate directory.

11.4 Extending Python with MPI

There are several Python packages available that purport to wrap MPI as a library and allow MPI functions to be called from Python.

These include

- pyMPI
- maroonmpi
- mpi4py
- myMPI
- Pypar

All of these except pyMPI work by wrapping the MPI library (which must be available on your system as a shared library, as discussed above), and exposing (some portion of) its interface to your Python script. This means they cannot be used interactively in parallel, since they do not address the issue of interactive input to multiple instances of Python running on different processors. The one exception is pyMPI, which alters the Python interpreter to address this issue, and (I believe) creates a new alternate executable (in place of python itself) as a result.

In principle any of these Python/MPI packages should work to invoke both calls to DSMC and MPI itself from a Python script running in parallel. However, when I downloaded and looked at a few of them, their docuemtation was incomplete and I had trouble with their installation. It's not clear if some of the packages are still being actively developed and supported.

The one I recommend, since I have successfully used it with DSMC, is Pypar. Pypar requires the ubiquitous Numpy package be installed in your Python. After launching python, type

```
>>> import numpy
```

to see if it is installed. If not, here is how to install it (version 1.3.0b1 as of April 2009). Unpack the numpy tarball and from its top−level directory, type

```
python setup.py build
sudo python setup.py install
```

The "sudo" is only needed if required to copy Numpy files into your Python distribution's site−packages directory.

To install Pypar (version pypar−2.1.0_66 as of April 2009), unpack it and from its "source" directory, type

```
python setup.py build
```

```
sudo python setup.py install
```

Again, the "sudo" is only needed if required to copy PyPar files into your Python distribution's site−packages directory.

If you have successully installed Pypar, you should be able to run python serially and type

```
>>> import pypar
```

without error. You should also be able to run python in parallel on a simple test script

```
% mpirun −np 4 python test.script
```

where test.script contains the lines

```
import pypar
print "Proc %d out of %d procs" % (pypar.rank(),pypar.size())
```

and see one line of output for each processor you ran on.

11.5 Testing the Python−DSMC interface

Before using DSMC in a Python program, one more step is needed. The interface to DSMC is via the Python ctypes package, which loads the shared DSMC library via a CDLL() call, which in turn is a wrapper on the C−library dlopen(). This command is different than a normal Python "import" and needs to be able to find the DSMC shared library, which is either in the Python site−packages directory or in a local directory you specified in the "python setup.py install" command, as described above.

The simplest way to do this is add a line like this to your .cshrc or other shell start−up file.

```
setenv LD_LIBRARY_PATH
${LD_LIBRARY_PATH}:/usr/local/lib/python2.5/site−packages
```

and then execute the shell file to insure the path has been updated. This will extend the path that dlopen() uses to look for shared libraries.

To test if the serial DSMC library has been successfully installed (mode 1 above), launch Python and type

```
>>> from lammps import lammps
>>> lmp = lammps()
```

If you get no errors, you're ready to use serial DSMC from Python.

If you built DSMC for parallel use (mode 2 or 3 above), launch Python in parallel:

```
% mpirun −np 4 python test.script
```

where test.script contains the lines

```
import pypar
from lammps import lammps
lmp = lammps()
print "Proc %d out of %d procs has" % (pypar.rank(),pypar.size()), lmp
pypar.finalize()
```

Again, if you get no errors, you're good to go.

Note that if you left out the "import pypar" line from this script, you would instantiate and run DSMC independently on each of the P processors specified in the mpirun command. You can test if Pypar is enabling true parallel Python and DSMC by adding a line to the above sequence of commands like lmp.file("in.lj") to run an input script and see if the DSMC run says it ran on P processors or if you get output from P duplicated 1−processor runs written to the screen. In the latter case, Pypar is not working correctly.

Note that this line:

```
from lammps import lammps
```

will import either the serial or parallel version of the DSMC library, as wrapped by lammps.py. But if you installed both via setup_serial.py and setup.py, it will always import the parallel version, since it attempts that first.

Note that if your Python script imports the Pypar package (as above), so that it can use MPI calls directly, then Pypar initializes MPI for you. Thus the last line of your Python script should be pypar.finalize(), to insure MPI is shut down correctly.

Also note that a Python script can be invoked in one of several ways:

% python foo.script % python −i foo.script % foo.script

The last command requires that the first line of the script be something like this:

#!/usr/local/bin/python #!/usr/local/bin/python −i

where the path points to where you have Python installed, and that you have made the script file executable:

% chmod +x foo.script

Without the "−i" flag, Python will exit when the script finishes. With the "−i" flag, you will be left in the Python interpreter when the script finishes, so you can type subsequent commands. As mentioned above, you can only run Python interactively when running Python on a single processor, not in parallel.

11.6 Using DSMC from Python

The Python interface to DSMC consists of a Python "lammps" module, the source code for which is in python/lammps.py, which creates a "lammps" object, with a set of methods that can be invoked on that object. The sample Python code below assumes you have first imported the "lammps" module in your Python script and its settings as follows:

```
from lammps import lammps
from lammps import LMPINT as INT
from lammps import LMPDOUBLE as DOUBLE
from lammps import LMPIPTR as IPTR
from lammps import LMPDPTR as DPTR
from lammps import LMPDPTRPTR as DPTRPTR
```

These are the methods defined by the lammps module. If you look at the file src/library.cpp you will see that they

correspond one−to−one with calls you can make to the DSMC library from a C++ or C or Fortran program.

```
lmp = lammps()           # create a DSMC object
lmp = lammps(list)       # ditto, with command−line args, list = ["−echo","screen"]

lmp.close()              # destroy a DSMC object

lmp.file(file)           # run an entire input script, file = "in.lj"
lmp.command(cmd)         # invoke a single DSMC command, cmd = "run 100"

xlo = lmp.extract_global(name,type)  # extract a global quantity
                                     # name = "boxxlo", "nlocal", etc
                                     # type = INT or DOUBLE

coords = lmp.extract_atom(name,type)      # extract a per−atom quantity
                                          # name = "x", "type", etc
                                          # type = IPTR or DPTR or DPTRPTR

eng = lmp.extract_compute(id,style,type)  # extract value(s) from a compute
v3 = lmp.extract_fix(id,style,type,i,j)   # extract value(s) from a fix
                                          # id = ID of compute or fix
                                          # style = 0 = global data
                                          #         1 = per−atom data
                                          #         2 = local data
                                          # type = 0 = scalar
                                          #        1 = vector
                                          #        2 = array
                                          # i,j = indices of value in global vector or array

var = lmp.extract_variable(name,group,flag)  # extract value(s) from a variable
                                             # name = name of variable
                                             # group = group ID (ignored for equal−style variables)
                                             # flag = 0 = equal−style variable
                                             #        1 = atom−style variable

natoms = lmp.get_natoms()                 # total # of atoms as int
x = lmp.get_coords()                      # return coords of all atoms in x
lmp.put_coords(x)                         # set all atom coords via x
```

The creation of a DSMC object does not take an MPI communicator as an argument. There should be a way to do this, so that the DSMC instance runs on a subset of processors, if desired, but I don't yet know how from Pypar. So for now, it runs on MPI_COMM_WORLD, which is all the processors.

The file() and command() methods allow an input script or single commands to be invoked.

The extract_global(), extract_atom(), extract_compute(), extract_fix(), and extract_variable() methods return values or pointers to data structures internal to DSMC.

For extract_global() see the src/library.cpp file for the list of valid names. New names could easily be added. A double or integer is returned. You need to specify the appropriate data type via the type argument.

For extract_atom(), a pointer to internal DSMC atom−based data is returned, which you can use via normal Python subscripting. See the extract() method in the src/atom.cpp file for a list of valid names. Again, new names could easily be added. A pointer to a vector of doubles or integers, or a pointer to an array of doubles (double **) is returned. You need to specify the appropriate data type via the type argument.

For extract_compute() and extract_fix(), the global, per−atom, or local data calulated by the compute or fix can be accessed. What is returned depends on whether the compute or fix calculates a scalar or vector or array. For a

scalar, a single double value is returned. If the compute or fix calculates a vector or array, a pointer to the internal DSMC data is returned, which you can use via normal Python subscripting. The one exception is that for a fix that calculates a global vector or array, a single double value from the vector or array is returned, indexed by I (vector) or I and J (array). I,J are zero−based indices. The I,J arguments can be left out if not needed. See this section of the manual for a discussion of global, per−atom, and local data, and of scalar, vector, and array data types. See the doc pages for individual computes and fixes for a description of what they calculate and store.

For extract_variable(), an equal−style or atom−style variable is evaluated and its result returned.

For equal−style variables a single double value is returned and the group argument is ignored. For atom−style variables, a vector of doubles is returned, one value per atom, which you can use via normal Python subscripting. The values will be zero for atoms not in the specified group.

The get_natoms() method returns the total number of atoms in the simulation, as an int. Note that extract_global("natoms") returns the same value, but as a double, which is the way DSMC stores it to allow for systems with more atoms than can be stored in an int (> 2 billion).

The get_coords() method returns an ctypes vector of doubles of length 3*natoms, for the coordinates of all the atoms in the simulation, ordered by x,y,z and then by atom ID (see code for put_coords() below). The array can be used via normal Python subscripting. If atom IDs are not consecutively ordered within DSMC, a None is returned as indication of an error.

Note that the data structure get_coords() returns is different from the data structure returned by extract_atom("x") in four ways. (1) Get_coords() returns a vector which you index as x[i]; extract_atom() returns an array which you index as x[i][j]. (2) Get_coords() orders the atoms by atom ID while extract_atom() does not. (3) Get_coords() returns a list of all atoms in the simulation; extract_atoms() returns just the atoms local to each processor. (4) Finally, the get_coords() data structure is a copy of the atom coords stored internally in DSMC, whereas extract_atom returns an array that points directly to the internal data. This means you can change values inside DSMC from Python by assigning a new values to the extract_atom() array. To do this with the get_atoms() vector, you need to change values in the vector, then invoke the put_coords() method.

The put_coords() method takes a vector of coordinates for all atoms in the simulation, assumed to be ordered by x,y,z and then by atom ID, and uses the values to overwrite the corresponding coordinates for each atom inside DSMC. This requires DSMC to have its "map" option enabled; see the atom_modify command for details. If it is not or if atom IDs are not consecutively ordered, no coordinates are reset,

The array of coordinates passed to put_coords() must be a ctypes vector of doubles, allocated and initialized something like this:

```
from ctypes import *
natoms = lmp.get_atoms()
n3 = 3*natoms
x = (c_double*n3)()
x0 = x coord of atom with ID 1
x1 = y coord of atom with ID 1
x2 = z coord of atom with ID 1
x3 = x coord of atom with ID 2
...
xn3-1 = z coord of atom with ID natoms
lmp.put_coords(x)
```

Alternatively, you can just change values in the vector returned by get_coords(), since it is a ctypes vector of doubles.

As noted above, these Python class methods correspond one−to−one with the functions in the DSMC library interface in src/library.cpp and library.h. This means you can extend the Python wrapper via the following steps:

- Add a new interface function to src/library.cpp and src/library.h.
- Verify the new function is syntactically correct by building DSMC as a library − see this section of the manual.
- Add a wrapper method in the Python DSMC module to python/lammps.py for this interface function.
- Rebuild the Python wrapper via python/setup_serial.py or python/setup.py.
- You should now be able to invoke the new interface function from a Python script. Isn't ctypes amazing?

11.7 Example Python scripts that use DSMC

These are the Python scripts included as demos in the python/examples directory of the DSMC distribution, to illustrate the kinds of things that are possible when Python wraps DSMC. If you create your own scripts, send them to us and we can include them in the DSMC distribution.

trivial.py	read/run a DSMC input script thru Python
demo.py	invoke various DSMC library interface routines
simple.py	mimic operation of couple/simple/simple.cpp in Python
gui.py	GUI go/stop/temperature−slider to control DSMC
plot.py	real−time temeperature plot with GnuPlot via Pizza.py
viz_tool.py	real−time viz via some viz package
vizplotgui_tool.py	combination of viz_tool.py and plot.py and gui.py

---

For the viz_tool.py and vizplotgui_tool.py commands, replace "tool" with "gl" or "atomeye" or "pymol" or "vmd", depending on what visualization package you have installed.

Note that for GL, you need to be able to run the Pizza.py GL tool, which is included in the pizza sub−directory. See the Pizza.py doc pages for more info:

Note that for AtomEye, you need version 3, and there is a line in the scripts that specifies the path and name of the executable. See the AtomEye WWW pages here or here for more details:

```
http://mt.seas.upenn.edu/Archive/Graphics/A
http://mt.seas.upenn.edu/Archive/Graphics/A3/A3.html
```

The latter link is to AtomEye 3 which has the scriping capability needed by these Python scripts.

Note that for PyMol, you need to have built and installed the open−source version of PyMol in your Python, so that you can import it from a Python script. See the PyMol WWW pages here or here for more details:

```
http://www.pymol.org
http://sourceforge.net/scm/?type=svn
```

The latter link is to the open−source version.

Note that for VMD, you need a fairly current version (1.8.7 works for me) and there are some lines in the pizza/vmd.py script for 4 PIZZA variables that have to match the VMD installation on your system.

See the python/README file for instructions on how to run them and the source code for individual scripts for comments about what they do.

Here are screenshots of the vizplotgui_tool.py script in action for different visualization package options. Click to see larger images:

....

12. Errors

This section describes the various kinds of errors you can encounter when using DSMC.

12.1 Common problems

If two DSMC runs do not produce the same answer on different machines or different numbers of processors, this is typically not a bug. In theory you should get identical answers on any number of processors and on any machine. In practice, numerical round−off can cause slight differences and eventual divergence of molecular dynamics phase space trajectories within a few 100s or few 1000s of timesteps. However, the statistical properties of the two runs (e.g. average energy or temperature) should still be the same.

If the velocity command is used to set initial atom velocities, a particular atom can be assigned a different velocity when the problem is run on a different number of processors or on different machines. If this happens, the phase space trajectories of the two simulations will rapidly diverge. See the discussion of the *loop* option in the velocity command for details and options that avoid this issue.

Similarly, the create_atoms command generates a lattice of atoms. For the same physical system, the ordering and numbering of atoms by atom ID may be different depending on the number of processors.

Some commands use random number generators which may be setup to produce different random number streams on each processor and hence will produce different effects when run on different numbers of processors. A commonly−used example is the fix langevin command for thermostatting.

A DSMC simulation typically has two stages, setup and run. Most DSMC errors are detected at setup time; others like a bond stretching too far may not occur until the middle of a run.

DSMC tries to flag errors and print informative error messages so you can fix the problem. Of course, DSMC cannot figure out your physics or numerical mistakes, like choosing too big a timestep, specifying erroneous force field coefficients, or putting 2 atoms on top of each other! If you run into errors that DSMC doesn't catch that you think it should flag, please send an email to the developers.

If you get an error message about an invalid command in your input script, you can determine what command is causing the problem by looking in the log.lammps file or using the echo command to see it on the screen. For a given command, DSMC expects certain arguments in a specified order. If you mess this up, DSMC will often flag the error, but it may read a bogus argument and assign a value that is valid, but not what you wanted. E.g. trying to read the string "abc" as an integer value and assigning the associated variable a value of 0.

Generally, DSMC will print a message to the screen and logfile and exit gracefully when it encounters a fatal error. Sometimes it will print a WARNING to the screen and logfile and continue on; you can decide if the WARNING is important or not. A WARNING message that is generated in the middle of a run is only printed to the screen, not to the logfile, to avoid cluttering up thermodynamic output. If DSMC crashes or hangs without spitting out an error message first then it could be a bug (see this section) or one of the following cases:

DSMC runs in the available memory a processor allows to be allocated. Most reasonable MD runs are compute limited, not memory limited, so this shouldn't be a bottleneck on most platforms. Almost all large memory allocations in the code are done via C−style malloc's which will generate an error message if you run out of memory. Smaller chunks of memory are allocated via C++ "new" statements. If you are unlucky you could run out of memory just when one of these small requests is made, in which case the code will crash or hang (in parallel), since DSMC doesn't trap on those errors.

Illegal arithmetic can cause DSMC to run slow or crash. This is typically due to invalid physics and numerics that your simulation is computing. If you see wild thermodynamic values or NaN values in your DSMC output, something is wrong with your simulation. If you suspect this is happening, it is a good idea to print out thermodynamic info frequently (e.g. every timestep) via the thermo so you can monitor what is happening. Visualizing the atom movement is also a good idea to insure your model is behaving as you expect.

In parallel, one way DSMC can hang is due to how different MPI implementations handle buffering of messages. If the code hangs without an error message, it may be that you need to specify an MPI setting or two (usually via an environment variable) to enable buffering or boost the sizes of messages that can be buffered.

12.2 Reporting bugs

If you are confident that you have found a bug in DSMC, follow these steps.

Check the New features and bug fixes section of the DSMC WWW site to see if the bug has already been reported or fixed or the Unfixed bug to see if a fix is pending.

Check the mailing list to see if it has been discussed before.

If not, send an email to the mailing list describing the problem with any ideas you have as to what is causing it or where in the code the problem might be. The developers will ask for more info if needed, such as an input script or data files.

The most useful thing you can do to help us fix the bug is to isolate the problem. Run it on the smallest number of atoms and fewest number of processors and with the simplest input script that reproduces the bug and try to identify what command or combination of commands is causing the problem.

As a last resort, you can send an email directly to the developers.

12.3 Error &warning messages

These are two alphabetic lists of the ERROR and WARNING messages DSMC prints out and the reason why. If the explanation here is not sufficient, the documentation for the offending command may help. Error and warning messages also list the source file and line number where the error was generated. For example, this message

ERROR: Illegal velocity command (velocity.cpp:78)

means that line #78 in the file src/velocity.cpp generated the error. Looking in the source code may help you figure out what went wrong.

Note that error messages from user−contributed packages are not listed here. If such an error occurs and is not self−explanatory, you'll need to look in the source code or contact the author of the package.

Errors:

1−3 bond count is inconsistent
: An inconsistency was detected when computing the number of 1−3 neighbors for each atom. This likely means something is wrong with the bond topologies you have defined.

1−4 bond count is inconsistent
: An inconsistency was detected when computing the number of 1−4 neighbors for each atom. This likely means something is wrong with the bond topologies you have defined.

Accelerated style in input script but no fix gpu
: GPU acceleration requires fix gpu in the input script.

All angle coeffs are not set
: All angle coefficients must be set in the data file or by the angle_coeff command before running a simulation.

All bond coeffs are not set
: All bond coefficients must be set in the data file or by the bond_coeff command before running a simulation.

All dihedral coeffs are not set
: All dihedral coefficients must be set in the data file or by the dihedral_coeff command before running a simulation.

All dipole moments are not set
: For atom styles that define dipole moments for each atom type, all moments must be set in the data file or by the dipole command before running a simulation.

All improper coeffs are not set
: All improper coefficients must be set in the data file or by the improper_coeff command before running a simulation.

All masses are not set
: For atom styles that define masses for each atom type, all masses must be set in the data file or by the mass command before running a simulation. They must also be set before using the velocity command.

All pair coeffs are not set
: All pair coefficients must be set in the data file or by the pair_coeff command before running a simulation.

All shapes are not set
: All atom types must have a shape setting, even if the particles are spherical.

All universe/uloop variables must have same # of values
: Self−explanatory.

All variables in next command must be same style
: Self−explanatory.

Angle atom missing in delete_bonds
: The delete_bonds command cannot find one or more atoms in a particular angle on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid angle.

Angle atom missing in set command
: The set command cannot find one or more atoms in a particular angle on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid angle.

Angle atoms %d %d %d missing on proc %d at step
: One or more of 3 atoms needed to compute a particular angle are missing on this processor. Typically this is because the pairwise cutoff is set too short or the angle has blown apart and an atom is too far away.

Angle coeff for hybrid has invalid style
: Angle style hybrid uses another angle style as one of its coefficients. The angle style used in the angle_coeff command or read from a restart file is not recognized.

Angle coeffs are not set
: No angle coefficients have been assigned in the data file or via the angle_coeff command.

Angle potential must be defined for SHAKE

When shaking angles, an angle_style potential must be used.

Angle style hybrid cannot have hybrid as an argument

Self−explanatory.

Angle style hybrid cannot have none as an argument

Self−explanatory.

Angle style hybrid cannot use same pair style twice

Self−explanatory.

Angle table must range from 0 to 180 degrees

Self−explanatory.

Angle table parameters did not set N

List of angle table parameters must include N setting.

Angle_coeff command before angle_style is defined

Coefficients cannot be set in the data file or via the angle_coeff command until an angle_style has been assigned.

Angle_coeff command before simulation box is defined

The angle_coeff command cannot be used before a read_data, read_restart, or create_box command.

Angle_coeff command when no angles allowed

The chosen atom style does not allow for angles to be defined.

Angle_style command when no angles allowed

The chosen atom style does not allow for angles to be defined.

Angles assigned incorrectly

Angles read in from the data file were not assigned correctly to atoms. This means there is something invalid about the topology definitions.

Angles defined but no angle types

The data file header lists angles but no angle types.

Another input script is already being processed

Cannot attempt to open a 2nd input script, when the original file is still being processed.

Arccos of invalid value in variable formula

Argument of arccos() must be between −1 and 1.

Arcsin of invalid value in variable formula

Argument of arcsin() must be between −1 and 1.

Atom IDs must be consecutive for velocity create loop all

Self−explanatory.

Atom count changed in fix neb

This is not allowed in a NEB calculation.

Atom count is inconsistent, cannot write restart file

Sum of atoms across processors does not equal initial total count. This is probably because you have lost some atoms.

Atom in too many rigid bodies − boost MAXBODY

Fix poems has a parameter MAXBODY (in fix_poems.cpp) which determines the maximum number of rigid bodies a single atom can belong to (i.e. a multibody joint). The bodies you have defined exceed this limit.

Atom sort did not operate correctly

This is an internal DSMC error. Please report it to the developers.

Atom sorting has bin size = 0.0

The neighbor cutoff is being used as the bin size, but it is zero. Thus you must explicitly list a bin size in the atom_modify sort command or turn off sorting.

Atom style hybrid cannot have hybrid as an argument

Self−explanatory.

Atom style hybrid cannot use same atom style twice

Self−explanatory.

Atom vector in equal−style variable formula

Atom vectors generate one value per atom which is not allowed in an equal−style variable.

Atom−style variable in equal−style variable formula

Atom−style variables generate one value per atom which is not allowed in an equal−style variable.

Atom_modify map command after simulation box is defined

The atom_modify map command cannot be used after a read_data, read_restart, or create_box command.

Atom_modify sort and first options cannot be used together

Self−explanatory.

Atom_style command after simulation box is defined

The atom_style command cannot be used after a read_data, read_restart, or create_box command.

Attempt to pop empty stack in fix box/relax

Internal DSMC error. Please report it to the developers.

Attempt to push beyond stack limit in fix box/relax

Internal DSMC error. Please report it to the developers.

Attempting to rescale a 0.0 temperature

Cannot rescale a temperature that is already 0.0.

Bad FENE bond

Two atoms in a FENE bond have become so far apart that the bond cannot be computed.

Bad TIP4P angle type for PPPM/TIP4P

Specified angle type is not valid.

Bad TIP4P bond type for PPPM/TIP4P

Specified bond type is not valid.

Bad grid of processors

The 3d grid of processors defined by the processors command does not match the number of processors DSMC is being run on.

Bad kspace_modify slab parameter

Kspace_modify value for the slab/volume keyword must be >= 2.0.

Bad principal moments

Fix rigid did not compute the principal moments of inertia of a rigid group of atoms correctly.

Bias compute does not calculate a velocity bias

The specified compute must compute a bias for temperature.

Bias compute does not calculate temperature

The specified compute must compute temperature.

Bias compute group does not match compute group

The specified compute must operate on the same group as the parent compute.

Big particle in fix srd cannot be point particle

Big particles must be extended spheriods or ellipsoids.

Bigint setting in lmptype.h is invalid

Size of bigint is less than size of tagint.

Bigint setting in lmptype.h is not compatible

Bigint stored in restart file is not consistent with DSMC version you are running.

Bitmapped lookup tables require int/float be same size

Cannot use pair tables on this machine, because of word sizes. Use the pair_modify command with table 0 instead.

Bitmapped table in file does not match requested table

Setting for bitmapped table in pair_coeff command must match table in file exactly.

Bitmapped table is incorrect length in table file

Number of table entries is not a correct power of 2.

Bond and angle potentials must be defined for TIP4P

Cannot use TIP4P pair potential unless bond and angle potentials are defined.

Bond atom missing in delete_bonds

The delete_bonds command cannot find one or more atoms in a particular bond on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid bond.

Bond atom missing in set command
: The set command cannot find one or more atoms in a particular bond on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid bond.

Bond atoms %d %d missing on proc %d at step
: One or both of 2 atoms needed to compute a particular bond are missing on this processor. Typically this is because the pairwise cutoff is set too short or the bond has blown apart and an atom is too far away.

Bond coeff for hybrid has invalid style
: Bond style hybrid uses another bond style as one of its coefficients. The bond style used in the bond_coeff command or read from a restart file is not recognized.

Bond coeffs are not set
: No bond coefficients have been assigned in the data file or via the bond_coeff command.

Bond potential must be defined for SHAKE
: Cannot use fix shake unless bond potential is defined.

Bond style hybrid cannot have hybrid as an argument
: Self−explanatory.

Bond style hybrid cannot have none as an argument
: Self−explanatory.

Bond style hybrid cannot use same pair style twice
: Self−explanatory.

Bond style quartic cannot be used with 3,4−body interactions
: No angle, dihedral, or improper styles can be defined when using bond style quartic.

Bond style quartic requires special_bonds = 1,1,1
: This is a restriction of the current bond quartic implementation.

Bond table parameters did not set N
: List of bond table parameters must include N setting.

Bond table values are not increasing
: The values in the tabulated file must be monotonically increasing.

Bond_coeff command before bond_style is defined
: Coefficients cannot be set in the data file or via the bond_coeff command until an bond_style has been assigned.

Bond_coeff command before simulation box is defined
: The bond_coeff command cannot be used before a read_data, read_restart, or create_box command.

Bond_coeff command when no bonds allowed
: The chosen atom style does not allow for bonds to be defined.

Bond_style command when no bonds allowed
: The chosen atom style does not allow for bonds to be defined.

Bonds assigned incorrectly
: Bonds read in from the data file were not assigned correctly to atoms. This means there is something invalid about the topology definitions.

Bonds defined but no bond types
: The data file header lists bonds but no bond types.

Both sides of boundary must be periodic
: Cannot specify a boundary as periodic only on the lo or hi side. Must be periodic on both sides.

Boundary command after simulation box is defined
: The boundary command cannot be used after a read_data, read_restart, or create_box command.

Box bounds are invalid
: The box boundaries specified in the read_data file are invalid. The lo value must be less than the hi value for all 3 dimensions.

Can not specify Pxy/Pxz/Pyz in fix box/relax with non−triclinic box
: Only triclinic boxes can be used with off−diagonal pressure components. See the region prism command for details.

Can not specify Pxy/Pxz/Pyz in fix nvt/npt/nph with non−triclinic box

Only triclinic boxes can be used with off−diagonal pressure components. See the region prism command for details.

Can only use NEB with 1−processor replicas

This is current restriction for NEB as implemented in DSMC.

Can only use TAD with 1−processor replicas for NEB

This is current restriction for NEB as implemented in DSMC.

Cannot (yet) use PPPM with triclinic box

This feature is not yet supported.

Cannot add atoms to fix move variable

Atoms can not be added afterwards to this fix option.

Cannot change box to orthogonal when tilt is non−zero

Self−explanatory

Cannot change box with certain fixes defined

The change_box command cannot be used when fix ave/spatial or fix/deform are defined .

Cannot change box with dumps defined

Self−explanatory.

Cannot change dump_modify every for dump dcd

The frequency of writing dump dcd snapshots cannot be changed.

Cannot change dump_modify every for dump xtc

The frequency of writing dump xtc snapshots cannot be changed.

Cannot change timestep once fix srd is setup

This is because various SRD properties depend on the timestep size.

Cannot change timestep with fix pour

This fix pre−computes some values based on the timestep, so it cannot be changed during a simulation run.

Cannot compute PPPM G

DSMC failed to compute a valid approximation for the PPPM g_ewald factor that partitions the computation between real space and k−space.

Cannot create an atom map unless atoms have IDs

The simulation requires a mapping from global atom IDs to local atoms, but the atoms that have been defined have no IDs.

Cannot create atoms with undefined lattice

Must use the lattice command before using the create_atoms command.

Cannot create/grow a vector/array of pointers for %s

DSMC code is making an illegal call to the templated memory allocaters, to create a vector or array of pointers.

Cannot create_atoms after reading restart file with per−atom info

The per−atom info was stored to be used when by a fix that you may re−define. If you add atoms before re−defining the fix, then there will not be a correct amount of per−atom info.

Cannot create_box after simulation box is defined

The create_box command cannot be used after a read_data, read_restart, or create_box command.

Cannot currently use pair reax with pair hybrid

This is not yet supported.

Cannot delete group all

Self−explanatory.

Cannot delete group currently used by a compute

Self−explanatory.

Cannot delete group currently used by a dump

Self−explanatory.

Cannot delete group currently used by a fix

Self−explanatory.

Cannot delete group currently used by atom_modify first

Self−explanatory.

Cannot displace_atoms after reading restart file with per−atom info

This is because the restart file info cannot be migrated with the atoms. You can get around this by performing a 0−timestep run which will assign the restart file info to actual atoms.

Cannot displace_box after reading restart file with per−atom info

This is because the restart file info cannot be migrated with the atoms. You can get around this by performing a 0−timestep run which will assign the restart file info to actual atoms.

Cannot displace_box on a non−periodic boundary

Self−explanatory.

Cannot dump sort on atom IDs with no atom IDs defined

Self−explanatory.

Cannot evaporate atoms in atom_modify first group

This is a restriction due to the way atoms are organized in a list to enable the atom_modify first command.

Cannot find delete_bonds group ID

Group ID used in the delete_bonds command does not exist.

Cannot have both pair_modify shift and tail set to yes

These 2 options are contradictory.

Cannot open AIREBO potential file %s

The specified AIREBO potential file cannot be opened. Check that the path and name are correct.

Cannot open COMB potential file %s

The specified COMB potential file cannot be opened. Check that the path and name are correct.

Cannot open EAM potential file %s

The specified EAM potential file cannot be opened. Check that the path and name are correct.

Cannot open EIM potential file %s

The specified EIM potential file cannot be opened. Check that the path and name are correct.

Cannot open MEAM potential file %s

The specified MEAM potential file cannot be opened. Check that the path and name are correct.

Cannot open Stillinger−Weber potential file %s

The specified SW potential file cannot be opened. Check that the path and name are correct.

Cannot open Tersoff potential file %s

The specified Tersoff potential file cannot be opened. Check that the path and name are correct.

Cannot open dir to search for restart file

Using a "*" in the name of the restart file will open the current directory to search for matching file names.

Cannot open dump file

The output file for the dump command cannot be opened. Check that the path and name are correct.

Cannot open file %s

The specified file cannot be opened. Check that the path and name are correct.

Cannot open fix ave/correlate file %s

The specified file cannot be opened. Check that the path and name are correct.

Cannot open fix ave/histo file %s

The specified file cannot be opened. Check that the path and name are correct.

Cannot open fix ave/spatial file %s

The specified file cannot be opened. Check that the path and name are correct.

Cannot open fix ave/time file %s

The specified file cannot be opened. Check that the path and name are correct.

Cannot open fix poems file %s

The specified file cannot be opened. Check that the path and name are correct.

Cannot open fix print file %s

The output file generated by the fix print command cannot be opened

Cannot open fix qeq/comb file %s

The output file for the fix qeq/combs command cannot be opened. Check that the path and name are correct.

Cannot open fix reax/bonds file %s

The output file for the fix reax/bonds command cannot be opened. Check that the path and name are correct.

Cannot open fix tmd file %s

The output file for the fix tmd command cannot be opened. Check that the path and name are correct.

Cannot open fix ttm file %s

The output file for the fix ttm command cannot be opened. Check that the path and name are correct.

Cannot open gzipped file

DSMC is attempting to open a gzipped version of the specified file but was unsuccessful. Check that the path and name are correct.

Cannot open input script %s

Self−explanatory.

Cannot open log.lammps

The default DSMC log file cannot be opened. Check that the directory you are running in allows for files to be created.

Cannot open logfile %s

The DSMC log file specified in the input script cannot be opened. Check that the path and name are correct.

Cannot open logfile

The DSMC log file named in a command−line argument cannot be opened. Check that the path and name are correct.

Cannot open pair_write file

The specified output file for pair energies and forces cannot be opened. Check that the path and name are correct.

Cannot open restart file %s

Self−explanatory.

Cannot open screen file

The screen file specified as a command−line argument cannot be opened. Check that the directory you are running in allows for files to be created.

Cannot open universe log file

For a multi−partition run, the master log file cannot be opened. Check that the directory you are running in allows for files to be created.

Cannot open universe screen file

For a multi−partition run, the master screen file cannot be opened. Check that the directory you are running in allows for files to be created.

Cannot read_data after simulation box is defined

The read_data command cannot be used after a read_data, read_restart, or create_box command.

Cannot read_restart after simulation box is defined

The read_restart command cannot be used after a read_data, read_restart, or create_box command.

Cannot redefine variable as a different style

An equal−style variable can be re−defined but only if it was originally an equal−style variable.

Cannot replicate 2d simulation in z dimension

The replicate command cannot replicate a 2d simulation in the z dimension.

Cannot replicate with fixes that store atom quantities

Either fixes are defined that create and store atom−based vectors or a restart file was read which included atom−based vectors for fixes. The replicate command cannot duplicate that information for new atoms. You should use the replicate command before fixes are applied to the system.

Cannot reset timestep with a dynamic region defined

Dynamic regions (see the region command) have a time dependence. Thus you cannot change the timestep when one or more of these are defined.

Cannot reset timestep with a time−dependent fix defined
: You cannot reset the timestep when a fix that keeps track of elapsed time is in place.

Cannot reset timestep with dump file already written to
: Changing the timestep will confuse when a dump file is written. Use the undump command, then restart the dump file.

Cannot reset timestep with restart file already written
: Changing the timestep will confuse when a restart file is written. Use the "restart 0" command to turn off restarts, then start them again.

Cannot restart fix rigid/nvt with different # of chains
: This is because the restart file contains per−chain info.

Cannot run 2d simulation with nonperiodic Z dimension
: Use the boundary command to make the z dimension periodic in order to run a 2d simulation.

Cannot set both respa pair and inner/middle/outer
: In the rRESPA integrator, you must compute pairwise potentials either all together (pair), or in pieces (inner/middle/outer). You can't do both.

Cannot set dipole for this atom style
: This atom style does not support dipole settings for each atom type.

Cannot set dump_modify flush for dump xtc
: Self−explanatory.

Cannot set mass for this atom style
: This atom style does not support mass settings for each atom type. Instead they are defined on a per−atom basis in the data file.

Cannot set non−zero image flag for non−periodic dimension
: Self−explanatory.

Cannot set non−zero z velocity for 2d simulation
: Self−explanatory.

Cannot set respa middle without inner/outer
: In the rRESPA integrator, you must define both a inner and outer setting in order to use a middle setting.

Cannot set shape for this atom style
: The atom style does not support this setting.

Cannot set this attribute for this atom style
: The attribute being set does not exist for the defined atom style.

Cannot set variable z velocity for 2d simulation
: Self−explanatory.

Cannot skew triclinic box in z for 2d simulation
: Self−explanatory.

Cannot use Ewald with 2d simulation
: The kspace style ewald cannot be used in 2d simulations. You can use 2d Ewald in a 3d simulation; see the kspace_modify command.

Cannot use Ewald with triclinic box
: This feature is not yet supported.

Cannot use NEB unless atom map exists
: Use the atom_modify command to create an atom map.

Cannot use NEB with a single replica
: Self−explanatory.

Cannot use NEB with atom_modify sort enabled
: This is current restriction for NEB implemented in DSMC.

Cannot use PPPM with 2d simulation
: The kspace style pppm cannot be used in 2d simulations. You can use 2d PPPM in a 3d simulation; see the kspace_modify command.

Cannot use PRD with a time−dependent fix defined
: PRD alters the timestep in ways that will mess up these fixes.

Cannot use PRD with a time−dependent region defined
PRD alters the timestep in ways that will mess up these regions.
Cannot use PRD with atom_modify sort enabled
This is a current restriction of PRD. You must turn off sorting, which is enabled by default, via the atom_modify command.
Cannot use PRD with multi−processor replicas unless atom map exists
Use the atom_modify command to create an atom map.
Cannot use TAD unless atom map exists for NEB
See atom_modify map command to set this.
Cannot use TAD with a single replica for NEB
NEB requires multiple replicas.
Cannot use TAD with atom_modify sort enabled for NEB
This is a current restriction of NEB.
Cannot use a damped dynamics min style with fix box/relax
This is a current restriction in DSMC. Use another minimizer style.
Cannot use a damped dynamics min style with per−atom DOF
This is a current restriction in DSMC. Use another minimizer style.
Cannot use compute cluster/atom unless atoms have IDs
Atom IDs are used to identify clusters.
Cannot use cwiggle in variable formula between runs
This is a function of elapsed time.
Cannot use delete_atoms unless atoms have IDs
Your atoms do not have IDs, so the delete_atoms command cannot be used.
Cannot use delete_bonds with non−molecular system
Your choice of atom style does not have bonds.
Cannot use fix TMD unless atom map exists
Using this fix requires the ability to lookup an atom index, which is provided by an atom map. An atom map does not exist (by default) for non−molecular problems. Using the atom_modify map command will force an atom map to be created.
Cannot use fix ave/spatial z for 2 dimensional model
Self−explanatory.
Cannot use fix bond/break with non−molecular systems
Self−explanatory.
Cannot use fix bond/create with non−molecular systems
Self−explanatory.
Cannot use fix box/relax on a 2nd non−periodic dimension
When specifying an off−diagonal pressure component, the 2nd of the two dimensions must be periodic. E.g. if the xy component is specified, then the y dimension must be periodic.
Cannot use fix box/relax on a non−periodic dimension
When specifying a diagonal pressure component, the dimension must be periodic.
Cannot use fix deform on a 2nd non−periodic boundary
When specifying a tilt factor change, the 2nd of the two dimensions must be periodic. E.g. if the xy tilt is specified, then the y dimension must be periodic.
Cannot use fix deform on a non−periodic boundary
When specifying a change is a box dimension, the dimension must be periodic.
Cannot use fix deform trate on a box with zero tilt
The trate style alters the current strain.
Cannot use fix enforce2d with 3d simulation
Self−explanatory.
Cannot use fix msst without per−type mass defined
Self−explanatory.
Cannot use fix npt and fix deform on same component of stress tensor

This would be changing the same box dimension twice.

Cannot use fix nvt/npt/nph on a 2nd non−periodic dimension

When specifying an off−diagonal pressure component, the 2nd of the two dimensions must be periodic. E.g. if the xy component is specified, then the y dimension must be periodic.

Cannot use fix nvt/npt/nph on a non−periodic dimension

When specifying a diagonal pressure component, the dimension must be periodic.

Cannot use fix pour with triclinic box

This feature is not yet supported.

Cannot use fix press/berendsen and fix deform on same component of stress tensor

These commands both change the box size/shape, so you cannot use both together.

Cannot use fix press/berendsen on a non−periodic dimension

Self−explanatory.

Cannot use fix press/berendsen with triclinic box

Self−explanatory.

Cannot use fix reax/bonds without pair_style reax

Self−explantory.

Cannot use fix shake with non−molecular system

Your choice of atom style does not have bonds.

Cannot use fix ttm with 2d simulation

This is a current restriction of this fix due to the grid it creates.

Cannot use fix ttm with triclinic box

This is a current restriction of this fix due to the grid it creates.

Cannot use fix wall in periodic dimension

Self−explanatory.

Cannot use fix wall zlo/zhi for a 2d simulation

Self−explanatory.

Cannot use fix wall/reflect in periodic dimension

Self−explanatory.

Cannot use fix wall/reflect zlo/zhi for a 2d simulation

Self−explanatory.

Cannot use fix wall/srd in periodic dimension

Self−explanatory.

Cannot use fix wall/srd more than once

Nor is their a need to since multiple walls can be specified in one command.

Cannot use fix wall/srd without fix srd

Self−explanatory.

Cannot use fix wall/srd zlo/zhi for a 2d simulation

Self−explanatory.

Cannot use force/neigh with triclinic box

This is a current limitation of the GPU implementation in DSMC.

Cannot use kspace solver on system with no charge

No atoms in system have a non−zero charge.

Cannot use neighbor bins − box size << cutoff

Too many neighbor bins will be created. This typically happens when the simulation box is very small in some dimension, compared to the neighbor cutoff. Use the "nsq" style instead of "bin" style.

Cannot use newton pair with GPU CHARMM pair style

See the newton command to change the setting.

Cannot use newton pair with GPU Gay−Berne pair style

See the newton command to change the setting.

Cannot use newton pair with GPU LJ pair style

See the newton command to change the setting.

Cannot use newton pair with GPU LJ96 pair style

See the newton command to change the setting.

Cannot use non−zero forces in an energy minimization

Fix setforce cannot be used in this manner. Use fix addforce instead.

Cannot use nonperiodic boundares with fix ttm

This fix requires a fully periodic simulation box.

Cannot use nonperiodic boundaries with Ewald

For kspace style ewald, all 3 dimensions must have periodic boundaries unless you use the kspace_modify command to define a 2d slab with a non−periodic z dimension.

Cannot use nonperiodic boundaries with PPPM

For kspace style pppm, all 3 dimensions must have periodic boundaries unless you use the kspace_modify command to define a 2d slab with a non−periodic z dimension.

Cannot use pair hybrid with GPU neighbor builds

See documentation for fix gpu.

Cannot use pair tail corrections with 2d simulations

The correction factors are only currently defined for 3d systems.

Cannot use ramp in variable formula between runs

This is because the ramp() function is time dependent.

Cannot use region INF or EDGE when box does not exist

Regions that extend to the box boundaries can only be used after the create_box command has been used.

Cannot use set atom with no atom IDs defined

Atom IDs are not defined, so they cannot be used to identify an atom.

Cannot use swiggle in variable formula between runs

This is a function of elapsed time.

Cannot use variable energy with constant force in fix addforce

This is because for constant force, DSMC can compute the change in energy directly.

Cannot use variable every setting for dump dcd

The format of DCD dump files requires snapshots be output at a constant frequency.

Cannot use variable every setting for dump xtc

The format of this file requires snapshots at regular intervals.

Cannot use vdisplace in variable formula between runs

This is a function of elapsed time.

Cannot use velocity create loop all unless atoms have IDs

Atoms in the simulation to do not have IDs, so this style of velocity creation cannot be performed.

Cannot use wall in periodic dimension

Self−explanatory.

Cannot wiggle and shear fix wall/gran

Cannot specify both options at the same time.

Cannot zero momentum of 0 atoms

The collection of atoms for which momentum is being computed has no atoms.

Change_box command before simulation box is defined

Self−explanatory.

Change_box operation is invalid

Cannot change orthogonal box to orthogonal or a triclinic box to triclinic.

Communicate group != atom_modify first group

Self−explanatory.

Compute ID for compute atom/molecule does not exist

Self−explanatory.

Compute ID for compute reduce does not exist

Self−explanatory.

Compute ID for fix ave/atom does not exist

Self−explanatory.

Compute ID for fix ave/correlate does not exist

Self−explanatory.

Compute ID for fix ave/histo does not exist

Self−explanatory.

Compute ID for fix ave/spatial does not exist

Self−explanatory.

Compute ID for fix ave/time does not exist

Self−explanatory.

Compute ID for fix store/state does not exist

Self−explanatory.

Compute ID must be alphanumeric or underscore characters

Self−explanatory.

Compute angle/local used when angles are not allowed

The atom style does not support angles.

Compute atom/molecule compute array is accessed out−of−range

Self−explanatory.

Compute atom/molecule compute does not calculate a per−atom array

Self−explanatory.

Compute atom/molecule compute does not calculate a per−atom vector

Self−explanatory.

Compute atom/molecule compute does not calculate per−atom values

Self−explanatory.

Compute atom/molecule fix array is accessed out−of−range

Self−explanatory.

Compute atom/molecule fix does not calculate a per−atom array

Self−explanatory.

Compute atom/molecule fix does not calculate a per−atom vector

Self−explanatory.

Compute atom/molecule fix does not calculate per−atom values

Self−explanatory.

Compute atom/molecule requires molecular atom style

Self−explanatory.

Compute atom/molecule variable is not atom−style variable

Self−explanatory.

Compute bond/local used when bonds are not allowed

The atom style does not support bonds.

Compute centro/atom requires a pair style be defined

This is because the computation of the centro−symmetry values uses a pairwise neighbor list.

Compute cluster/atom cutoff is longer than pairwise cutoff

Cannot identify clusters beyond cutoff.

Compute cluster/atom requires a pair style be defined

This is so that the pair style defines a cutoff distance which is used to find clusters.

Compute cna/atom cutoff is longer than pairwise cutoff

Self−explantory.

Compute cna/atom requires a pair style be defined

Self−explantory.

Compute com/molecule requires molecular atom style

Self−explanatory.

Compute coord/atom cutoff is longer than pairwise cutoff

Cannot compute coordination at distances longer than the pair cutoff, since those atoms are not in the neighbor list.

Compute coord/atom requires a pair style be defined

Self−explantory.

Compute damage/atom requires peridynamic potential
: Damage is a Peridynamic−specific metric. It requires you to be running a Peridynamics simulation.

Compute dihedral/local used when dihedrals are not allowed
: The atom style does not support dihedrals.

Compute does not allow an extra compute or fix to be reset
: This is an internal DSMC error. Please report it to the developers.

Compute erotate/asphere cannot be used with atom attributes diameter or rmass
: These attributes override the shape and mass settings, so cannot be used.

Compute erotate/asphere requires atom attributes angmom, quat, shape
: An atom style that defines these attributes must be used.

Compute erotate/asphere requires extended particles
: This compute cannot be used with point paritlces.

Compute erotate/sphere requires atom attribute omega
: An atom style that defines this attribute must be used.

Compute erotate/sphere requires atom attribute radius or shape
: An atom style that defines these attributes must be used.

Compute erotate/sphere requires spherical particle shapes
: Self−explanatory.

Compute event/displace has invalid fix event assigned
: This is an internal DSMC error. Please report it to the developers.

Compute group/group group ID does not exist
: Self−explanatory.

Compute gyration/molecule requires molecular atom style
: Self−explanatory.

Compute heat/flux compute ID does not compute ke/atom
: Self−explanatory.

Compute heat/flux compute ID does not compute pe/atom
: Self−explanatory.

Compute heat/flux compute ID does not compute stress/atom
: Self−explanatory.

Compute improper/local used when impropers are not allowed
: The atom style does not support impropers.

Compute msd/molecule requires molecular atom style
: Self−explanatory.

Compute pair must use group all
: Pair styles accumlate energy on all atoms.

Compute pe must use group all
: Energies computed by potentials (pair, bond, etc) are computed on all atoms.

Compute pressure must use group all
: Virial contributions computed by potentials (pair, bond, etc) are computed on all atoms.

Compute pressure temperature ID does not compute temperature
: The compute ID assigned to a pressure computation must compute temperature.

Compute property/atom for atom property that isn't allocated
: Self−explanatory.

Compute property/local cannot use these inputs together
: Only inputs that generate the same number of datums can be used togther. E.g. bond and angle quantities cannot be mixed.

Compute property/local for property that isn't allocated
: Self−explanatory.

Compute property/molecule requires molecular atom style
: Self−explanatory.

Compute rdf requires a pair style be defined

Self−explanatory.

Compute reduce compute array is accessed out−of−range

Self−explanatory.

Compute reduce compute calculates global values

A compute that calculates peratom or local values is required.

Compute reduce compute does not calculate a local array

Self−explanatory.

Compute reduce compute does not calculate a local vector

Self−explanatory.

Compute reduce compute does not calculate a per−atom array

Self−explanatory.

Compute reduce compute does not calculate a per−atom vector

Self−explanatory.

Compute reduce fix array is accessed out−of−range

Self−explanatory.

Compute reduce fix calculates global values

A fix that calculates peratom or local values is required.

Compute reduce fix does not calculate a local array

Self−explanatory.

Compute reduce fix does not calculate a local vector

Self−explanatory.

Compute reduce fix does not calculate a per−atom array

Self−explanatory.

Compute reduce fix does not calculate a per−atom vector

Self−explanatory.

Compute reduce replace requires min or max mode

Self−explanatory.

Compute reduce variable is not atom−style variable

Self−explanatory.

Compute temp/asphere cannot be used with atom attributes diameter or rmass

These attributes override the shape and mass settings, so cannot be used.

Compute temp/asphere requires atom attributes angmom, quat, shape

An atom style that defines these attributes must be used.

Compute temp/asphere requires extended particles

This compute cannot be used with point paritlces.

Compute temp/partial cannot use vz for 2d systemx

Self−explanatory.

Compute temp/profile cannot bin z for 2d systems

Self−explanatory.

Compute temp/profile cannot use vz for 2d systemx

Self−explanatory.

Compute temp/sphere requires atom attribute omega

An atom style that defines this attribute must be used.

Compute temp/sphere requires atom attribute radius or shape

An atom style that defines these attributes must be used.

Compute temp/sphere requires spherical particle shapes

Self−explanatory.

Compute ti kspace style does not exist

Self−explanatory.

Compute ti pair style does not exist

Self−explanatory.

Compute ti tail when pair style does not compute tail corrections

Self−explanatory.

Compute used in variable between runs is not current

Computes cannot be invoked by a variable in between runs. Thus they must have been evaluated on the last timestep of the previous run in order for their value(s) to be accessed. See the doc page for the variable command for more info.

Compute used in variable thermo keyword between runs is not current

Some thermo keywords rely on a compute to calculate their value(s). Computes cannot be invoked by a variable in between runs. Thus they must have been evaluated on the last timestep of the previous run in order for their value(s) to be accessed. See the doc page for the variable command for more info.

Computed temperature for fix temp/berendsen cannot be 0.0

Self−explanatory.

Computed temperature for fix temp/rescale cannot be 0.0

Cannot rescale the temperature to a new value if the current temperature is 0.0.

Could not count initial bonds in fix bond/create

Could not find one of the atoms in a bond on this processor.

Could not create 3d FFT plan

The FFT setup in pppm failed.

Could not create 3d remap plan

The FFT setup in pppm failed.

Could not find atom_modify first group ID

Self−explanatory.

Could not find compute ID for PRD

Self−explanatory.

Could not find compute ID for TAD

Self−explanatory.

Could not find compute ID for temperature bias

Self−explanatory.

Could not find compute ID to delete

Self−explanatory.

Could not find compute displace/atom fix ID

Self−explanatory.

Could not find compute event/displace fix ID

Self−explanatory.

Could not find compute group ID

Self−explanatory.

Could not find compute heat/flux compute ID

Self−explanatory.

Could not find compute msd fix ID

Self−explanatory.

Could not find compute pressure temperature ID

The compute ID for calculating temperature does not exist.

Could not find compute_modify ID

Self−explanatory.

Could not find delete_atoms group ID

Group ID used in the delete_atoms command does not exist.

Could not find delete_atoms region ID

Region ID used in the delete_atoms command does not exist.

Could not find displace_atoms group ID

Group ID used in the displace_atoms command does not exist.

Could not find displace_box group ID

Group ID used in the displace_box command does not exist.

Could not find dump cfg compute ID

Self−explanatory.

Could not find dump cfg fix ID

Self−explanatory.

Could not find dump cfg variable name

Self−explanatory.

Could not find dump custom compute ID

The compute ID needed by dump custom to compute a per−atom quantity does not exist.

Could not find dump custom fix ID

Self−explanatory.

Could not find dump custom variable name

Self−explanatory.

Could not find dump group ID

A group ID used in the dump command does not exist.

Could not find dump local compute ID

Self−explanatory.

Could not find dump local fix ID

Self−explanatory.

Could not find dump modify compute ID

Self−explanatory.

Could not find dump modify fix ID

Self−explanatory.

Could not find dump modify variable name

Self−explanatory.

Could not find fix ID to delete

Self−explanatory.

Could not find fix group ID

A group ID used in the fix command does not exist.

Could not find fix msst compute ID

Self−explanatory.

Could not find fix poems group ID

A group ID used in the fix poems command does not exist.

Could not find fix recenter group ID

A group ID used in the fix recenter command does not exist.

Could not find fix rigid group ID

A group ID used in the fix rigid command does not exist.

Could not find fix srd group ID

Self−explanatory.

Could not find fix_modify ID

A fix ID used in the fix_modify command does not exist.

Could not find fix_modify pressure ID

The compute ID for computing pressure does not exist.

Could not find fix_modify temperature ID

The compute ID for computing temperature does not exist.

Could not find group delete group ID

Self−explanatory.

Could not find/initialize a specified accelerator device

Your GPU setup is invalid.

Could not find set group ID

Group ID specified in set command does not exist.

Could not find thermo compute ID

Compute ID specified in thermo_style command does not exist.

Could not find thermo custom compute ID

The compute ID needed by thermo style custom to compute a requested quantity does not exist.

Could not find thermo custom fix ID

The fix ID needed by thermo style custom to compute a requested quantity does not exist.

Could not find thermo custom variable name

Self−explanatory.

Could not find thermo fix ID

Fix ID specified in thermo_style command does not exist.

Could not find thermo_modify pressure ID

The compute ID needed by thermo style custom to compute pressure does not exist.

Could not find thermo_modify temperature ID

The compute ID needed by thermo style custom to compute temperature does not exist.

Could not find undump ID

A dump ID used in the undump command does not exist.

Could not find velocity group ID

A group ID used in the velocity command does not exist.

Could not find velocity temperature ID

The compute ID needed by the velocity command to compute temperature does not exist.

Could not grab element entry from EIM potential file

Self−explanatory

Could not grab global entry from EIM potential file

Self−explanatory.

Could not grab pair entry from EIM potential file

Self−explanatory.

Could not set finite−size particle attribute in fix rigid

The particle has a finite size but its attributes could not be determined.

Coulomb cutoffs of pair hybrid sub−styles do not match

If using a Kspace solver, all Coulomb cutoffs of long pair styles must be the same.

Cound not find dump_modify ID

Self−explanatory.

Create_atoms command before simulation box is defined

The create_atoms command cannot be used before a read_data, read_restart, or create_box command.

Create_atoms region ID does not exist

A region ID used in the create_atoms command does not exist.

Create_box region ID does not exist

A region ID used in the create_box command does not exist.

Create_box region does not support a bounding box

Not all regions represent bounded volumes. You cannot use such a region with the create_box command.

Cyclic loop in joint connections

Fix poems cannot (yet) work with coupled bodies whose joints connect the bodies in a ring (or cycle).

Degenerate lattice primitive vectors

Invalid set of 3 lattice vectors for lattice command.

Delete region ID does not exist

Self−explanatory.

Delete_atoms command before simulation box is defined

The delete_atoms command cannot be used before a read_data, read_restart, or create_box command.

Delete_atoms cutoff > neighbor cutoff

Cannot delete atoms further away than a processor knows about.

Delete_atoms requires a pair style be defined

This is because atom deletion within a cutoff uses a pairwise neighbor list.

Delete_bonds command before simulation box is defined

The delete_bonds command cannot be used before a read_data, read_restart, or create_box command.

Delete_bonds command with no atoms existing

No atoms are yet defined so the delete_bonds command cannot be used.

Deposition region extends outside simulation box

Self−explanatory.

Did not assign all atoms correctly

Atoms read in from a data file were not assigned correctly to processors. This is likely due to some atom coordinates being outside a non−periodic simulation box.

Did not find all elements in MEAM library file

The requested elements were not found in the MEAM file.

Did not find fix shake partner info

Could not find bond partners implied by fix shake command. This error can be triggered if the delete_bonds command was used before fix shake, and it removed bonds without resetting the 1−2, 1−3, 1−4 weighting list via the special keyword.

Did not find keyword in table file

Keyword used in pair_coeff command was not found in table file.

Did not set temp for fix rigid/nvt

The temp keyword must be used.

Dihedral atom missing in delete_bonds

The delete_bonds command cannot find one or more atoms in a particular dihedral on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid dihedral.

Dihedral atom missing in set command

The set command cannot find one or more atoms in a particular dihedral on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid dihedral.

Dihedral atoms %d %d %d %d missing on proc %d at step

One or more of 4 atoms needed to compute a particular dihedral are missing on this processor. Typically this is because the pairwise cutoff is set too short or the dihedral has blown apart and an atom is too far away.

Dihedral charmm is incompatible with Pair style

Dihedral style charmm must be used with a pair style charmm in order for the 1−4 epsilon/sigma parameters to be defined.

Dihedral coeff for hybrid has invalid style

Dihedral style hybrid uses another dihedral style as one of its coefficients. The dihedral style used in the dihedral_coeff command or read from a restart file is not recognized.

Dihedral coeffs are not set

No dihedral coefficients have been assigned in the data file or via the dihedral_coeff command.

Dihedral style hybrid cannot have hybrid as an argument

Self−explanatory.

Dihedral style hybrid cannot have none as an argument

Self−explanatory.

Dihedral style hybrid cannot use same dihedral style twice

Self−explanatory.

Dihedral_coeff command before dihedral_style is defined

Coefficients cannot be set in the data file or via the dihedral_coeff command until an dihedral_style has been assigned.

Dihedral_coeff command before simulation box is defined

The dihedral_coeff command cannot be used before a read_data, read_restart, or create_box command.

Dihedral_coeff command when no dihedrals allowed

The chosen atom style does not allow for dihedrals to be defined.

Dihedral_style command when no dihedrals allowed

The chosen atom style does not allow for dihedrals to be defined.

Dihedrals assigned incorrectly

Dihedrals read in from the data file were not assigned correctly to atoms. This means there is something invalid about the topology definitions.

Dihedrals defined but no dihedral types
The data file header lists dihedrals but no dihedral types.
Dimension command after simulation box is defined
The dimension command cannot be used after a read_data, read_restart, or create_box command.
Dipole command before simulation box is defined
The dipole command cannot be used before a read_data, read_restart, or create_box command.
Displace_atoms command before simulation box is defined
The displace_atoms command cannot be used before a read_data, read_restart, or create_box command.
Displace_box command before simulation box is defined
Self−explanatory.
Displace_box tilt factors require triclinic box
Cannot use tilt factors unless the simulation box is non−orthogonal.
Distance must be > 0 for compute event/displace
Self−explanatory.
Divide by 0 in influence function of pair peri/lps
This should not normally occur. It is likely a problem with your model.
Divide by 0 in variable formula
Self−explanatory.
Domain too large for neighbor bins
The domain has become extremely large so that neighbor bins cannot be used. Most likely, one or more atoms have been blown out of the simulation box to a great distance.
Double precision is not supported on this accelerator.
In this case, you must compile the GPU library for single precision.
Dump cfg and fix not computed at compatible times
The fix must produce per−atom quantities on timesteps that dump cfg needs them.
Dump cfg arguments must start with 'id type xs ys zs'
This is a requirement of the CFG output format.
Dump cfg requires one snapshot per file
Use the wildcard "*" character in the filename.
Dump custom and fix not computed at compatible times
The fix must produce per−atom quantities on timesteps that dump custom needs them.
Dump custom compute does not calculate per−atom array
Self−explanatory.
Dump custom compute does not calculate per−atom vector
Self−explanatory.
Dump custom compute does not compute per−atom info
Self−explanatory.
Dump custom compute vector is accessed out−of−range
Self−explanatory.
Dump custom fix does not compute per−atom array
Self−explanatory.
Dump custom fix does not compute per−atom info
Self−explanatory.
Dump custom fix does not compute per−atom vector
Self−explanatory.
Dump custom fix vector is accessed out−of−range
Self−explanatory.
Dump custom variable is not atom−style variable
Only atom−style variables generate per−atom quantities, needed for dump output.
Dump dcd of non−matching # of atoms
Every snapshot written by dump dcd must contain the same # of atoms.
Dump dcd requires sorting by atom ID

Use the dump_modify sort command to enable this.

Dump every variable returned a bad timestep

The variable must return a timestep greater than the current timestep.

Dump local and fix not computed at compatible times

The fix must produce per−atom quantities on timesteps that dump local needs them.

Dump local attributes contain no compute or fix

Self−explanatory.

Dump local cannot sort by atom ID

This is because dump local does not really dump per−atom info.

Dump local compute does not calculate local array

Self−explanatory.

Dump local compute does not calculate local vector

Self−explanatory.

Dump local compute does not compute local info

Self−explanatory.

Dump local compute vector is accessed out−of−range

Self−explanatory.

Dump local count is not consistent across input fields

Every column of output must be the same length.

Dump local fix does not compute local array

Self−explanatory.

Dump local fix does not compute local info

Self−explanatory.

Dump local fix does not compute local vector

Self−explanatory.

Dump local fix vector is accessed out−of−range

Self−explanatory.

Dump modify compute ID does not compute per−atom array

Self−explanatory.

Dump modify compute ID does not compute per−atom info

Self−explanatory.

Dump modify compute ID does not compute per−atom vector

Self−explanatory.

Dump modify compute ID vector is not large enough

Self−explanatory.

Dump modify element names do not match atom types

Number of element names must equal number of atom types.

Dump modify fix ID does not compute per−atom array

Self−explanatory.

Dump modify fix ID does not compute per−atom info

Self−explanatory.

Dump modify fix ID does not compute per−atom vector

Self−explanatory.

Dump modify fix ID vector is not large enough

Self−explanatory.

Dump modify variable is not atom−style variable

Self−explanatory.

Dump sort column is invalid

Self−explanatory.

Dump xtc requires sorting by atom ID

Use the dump_modify sort command to enable this.

Dump_modify region ID does not exist

Self−explanatory.

Dumping an atom property that isn't allocated

The chosen atom style does not define the per−atom quantity being dumped.

Dumping an atom quantity that isn't allocated

Only per−atom quantities that are defined for the atom style being used are allowed.

Duplicate particle in PeriDynamic bond − simulation box is too small

This is likely because your box length is shorter than 2 times the bond length.

Electronic temperature dropped below zero

Something has gone wrong with the fix ttm electron temperature model.

Empty brackets in variable

There is no variable syntax that uses empty brackets. Check the variable doc page.

Energy was not tallied on needed timestep

You are using a thermo keyword that requires potentials to have tallied energy, but they didn't on this timestep. See the variable doc page for ideas on how to make this work.

Expected floating point parameter in input script or data file

The quantity being read is an integer on non−numeric value.

Expected floating point parameter in variable definition

The quantity being read is a non−numeric value.

Expected integer parameter in input script or data file

The quantity being read is a floating point or non−numeric value.

Expected integer parameter in variable definition

The quantity being read is a floating point or non−numeric value.

Failed to allocate %d bytes for array %s

Your DSMC simulation has run out of memory. You need to run a smaller simulation or on more processors.

Failed to reallocate %d bytes for array %s

Your DSMC simulation has run out of memory. You need to run a smaller simulation or on more processors.

Fewer SRD bins than processors in some dimension

This is not allowed. Make your SRD bin size smaller.

Final box dimension due to fix deform is < 0.0

Self−explanatory.

Fix gpu split must be positive for hybrid pair styles.

See documentation for fix gpu.

Fix ID for compute atom/molecule does not exist

Self−explanatory.

Fix ID for compute reduce does not exist

Self−explanatory.

Fix ID for fix ave/atom does not exist

Self−explanatory.

Fix ID for fix ave/correlate does not exist

Self−explanatory.

Fix ID for fix ave/histo does not exist

Self−explanatory.

Fix ID for fix ave/spatial does not exist

Self−explanatory.

Fix ID for fix ave/time does not exist

Self−explanatory.

Fix ID for fix store/state does not exist

Self−explanatory

Fix ID must be alphanumeric or underscore characters

Self−explanatory.

Fix SRD cannot have both atom attributes angmom and omega
Use either spheroid solute particles or fully generalized aspherical solute particles.
Fix SRD no−slip requires atom attribute torque
This is because the SRD collisions will impart torque to the solute particles.
Fix SRD requires atom attribute angmom or omega
This is because the SRD collisions with cause the solute particles to rotate.
Fix SRD requires atom attribute radius or shape
This is because the solute particles must be finite−size particles, not point particles.
Fix SRD: bad bin assignment for SRD advection
Something has gone wrong in your SRD model; try using more conservative settings.
Fix SRD: bad search bin assignment
Something has gone wrong in your SRD model; try using more conservative settings.
Fix SRD: bad stencil bin for big particle
Something has gone wrong in your SRD model; try using more conservative settings.
Fix SRD: too many big particles in bin
Reset the ATOMPERBIN parameter at the top of fix_srd.cpp to a larger value, and re−compile the code.
Fix SRD: too many walls in bin
This should not happen unless your system has been setup incorrectly.
Fix adapt kspace style does not exist
Self−explanatory.
Fix adapt pair style does not exist
Self−explanatory
Fix adapt pair style param not supported
The pair style does not know about the parameter you specified.
Fix adapt requires atom attribute diameter
The atom style being used does not specify an atom diameter.
Fix adapt type pair range is not valid for pair hybrid sub−style
Self−explanatory.
Fix ave/atom compute array is accessed out−of−range
Self−explanatory.
Fix ave/atom compute does not calculate a per−atom array
Self−explanatory.
Fix ave/atom compute does not calculate a per−atom vector
A compute used by fix ave/atom must generate per−atom values.
Fix ave/atom compute does not calculate per−atom values
A compute used by fix ave/atom must generate per−atom values.
Fix ave/atom fix array is accessed out−of−range
Self−explanatory.
Fix ave/atom fix does not calculate a per−atom array
Self−explanatory.
Fix ave/atom fix does not calculate a per−atom vector
A fix used by fix ave/atom must generate per−atom values.
Fix ave/atom fix does not calculate per−atom values
A fix used by fix ave/atom must generate per−atom values.
Fix ave/atom variable is not atom−style variable
A variable used by fix ave/atom must generate per−atom values.
Fix ave/histo cannot input local values in scalar mode
Self−explanatory.
Fix ave/histo cannot input per−atom values in scalar mode
Self−explanatory.
Fix ave/histo compute array is accessed out−of−range
Self−explanatory.

Fix ave/histo compute does not calculate a global array
Self−explanatory.
Fix ave/histo compute does not calculate a global scalar
Self−explanatory.
Fix ave/histo compute does not calculate a global vector
Self−explanatory.
Fix ave/histo compute does not calculate a local array
Self−explanatory.
Fix ave/histo compute does not calculate a local vector
Self−explanatory.
Fix ave/histo compute does not calculate a per−atom array
Self−explanatory.
Fix ave/histo compute does not calculate a per−atom vector
Self−explanatory.
Fix ave/histo compute does not calculate local values
Self−explanatory.
Fix ave/histo compute does not calculate per−atom values
Self−explanatory.
Fix ave/histo compute vector is accessed out−of−range
Self−explanatory.
Fix ave/histo fix array is accessed out−of−range
Self−explanatory.
Fix ave/histo fix does not calculate a global array
Self−explanatory.
Fix ave/histo fix does not calculate a global scalar
Self−explanatory.
Fix ave/histo fix does not calculate a global vector
Self−explanatory.
Fix ave/histo fix does not calculate a local array
Self−explanatory.
Fix ave/histo fix does not calculate a local vector
Self−explanatory.
Fix ave/histo fix does not calculate a per−atom array
Self−explanatory.
Fix ave/histo fix does not calculate a per−atom vector
Self−explanatory.
Fix ave/histo fix does not calculate local values
Self−explanatory.
Fix ave/histo fix does not calculate per−atom values
Self−explanatory.
Fix ave/histo fix vector is accessed out−of−range
Self−explanatory.
Fix ave/histo input is invalid compute
Self−explanatory.
Fix ave/histo input is invalid fix
Self−explanatory.
Fix ave/histo input is invalid variable
Self−explanatory.
Fix ave/histo inputs are not all global, peratom, or local
All inputs in a single fix ave/histo command must be of the same style.
Fix ave/spatial compute does not calculate a per−atom array
Self−explanatory.

Fix ave/spatial compute does not calculate a per−atom vector
A compute used by fix ave/spatial must generate per−atom values.
Fix ave/spatial compute does not calculate per−atom values
A compute used by fix ave/spatial must generate per−atom values.
Fix ave/spatial compute vector is accessed out−of−range
The index for the vector is out of bounds.
Fix ave/spatial fix does not calculate a per−atom array
Self−explanatory.
Fix ave/spatial fix does not calculate a per−atom vector
A fix used by fix ave/spatial must generate per−atom values.
Fix ave/spatial fix does not calculate per−atom values
A fix used by fix ave/spatial must generate per−atom values.
Fix ave/spatial fix vector is accessed out−of−range
The index for the vector is out of bounds.
Fix ave/spatial for triclinic boxes requires units reduced
Self−explanatory.
Fix ave/spatial settings invalid with changing box
If the ave setting is "running" or "window" and the box size/shape changes during the simulation, then the units setting must be "reduced", else the number of bins may change.
Fix ave/spatial variable is not atom−style variable
A variable used by fix ave/spatial must generate per−atom values.
Fix ave/time cannot set output array intensive/extensive from these inputs
One of more of the vector inputs has individual elements which are flagged as intensive or extensive. Such an input cannot be flagged as all intensive/extensive when turned into an array by fix ave/time.
Fix ave/time cannot use variable with vector mode
Variables produce scalar values.
Fix ave/time columns are inconsistent lengths
Self−explanatory.
Fix ave/time compute array is accessed out−of−range
Self−explanatory.
Fix ave/time compute does not calculate a scalar
Only computes that calculate a scalar or vector quantity (not a per−atom quantity) can be used with fix ave/time.
Fix ave/time compute does not calculate a vector
Only computes that calculate a scalar or vector quantity (not a per−atom quantity) can be used with fix ave/time.
Fix ave/time compute does not calculate an array
Self−explanatory.
Fix ave/time compute vector is accessed out−of−range
The index for the vector is out of bounds.
Fix ave/time fix array is accessed out−of−range
Self−explanatory.
Fix ave/time fix does not calculate a scalar
A fix used by fix ave/time must generate global values.
Fix ave/time fix does not calculate a vector
A fix used by fix ave/time must generate global values.
Fix ave/time fix does not calculate an array
Self−explanatory.
Fix ave/time fix vector is accessed out−of−range
The index for the vector is out of bounds.
Fix ave/time variable is not equal−style variable
A variable used by fix ave/time must generate a global value.

Fix bond/break requires special_bonds = 0,1,1
This is a restriction of the current fix bond/break implementation.
Fix bond/create cutoff is longer than pairwise cutoff
This is not allowed because bond creation is done using the pairwise neighbor list.
Fix bond/create requires special_bonds coul = 0,1,1
Self−explanatory.
Fix bond/create requires special_bonds lj = 0,1,1
Self−explanatory.
Fix bond/swap cannot use dihedral or improper styles
These styles cannot be defined when using this fix.
Fix bond/swap requires pair and bond styles
Self−explanatory.
Fix bond/swap requires special_bonds = 0,1,1
Self−explanatory.
Fix box/relax generated negative box length
The pressure being applied is likely too large. Try applying it incrementally, to build to the high pressure.
Fix command before simulation box is defined
The fix command cannot be used before a read_data, read_restart, or create_box command.
Fix deform is changing yz by too much with changing xy
When both yz and xy are changing, it induces changes in xz if the box must flip from one tilt extreme to another. Thus it is not allowed for yz to grow so much that a flip is induced.
Fix deform tilt factors require triclinic box
Cannot deform the tilt factors of a simulation box unless it is a triclinic (non−orthogonal) box.
Fix deform volume setting is invalid
Cannot use volume style unless other dimensions are being controlled.
Fix deposit region cannot be dynamic
Only static regions can be used with fix deposit.
Fix deposit region does not support a bounding box
Not all regions represent bounded volumes. You cannot use such a region with the fix deposit command.
Fix efield requires atom attribute q
Self−explanatory.
Fix evaporate molecule requires atom attribute molecule
The atom style being used does not define a molecule ID.
Fix external callback function not set
This must be done by an external program in order to use this fix.
Fix for fix ave/atom not computed at compatible time
Fixes generate their values on specific timesteps. Fix ave/atom is requesting a value on a non−allowed timestep.
Fix for fix ave/correlate not computed at compatible time
Fixes generate their values on specific timesteps. Fix ave/correlate is requesting a value on a non−allowed timestep.
Fix for fix ave/histo not computed at compatible time
Fixes generate their values on specific timesteps. Fix ave/histo is requesting a value on a non−allowed timestep.
Fix for fix ave/spatial not computed at compatible time
Fixes generate their values on specific timesteps. Fix ave/spatial is requesting a value on a non−allowed timestep.
Fix for fix ave/time not computed at compatible time
Fixes generate their values on specific timesteps. Fix ave/time is requesting a value on a non−allowed timestep.
Fix for fix store/state not computed at compatible time
Fixes generate their values on specific timesteps. Fix store/state is requesting a value on a non−allowed

timestep.

Fix freeze requires atom attribute torque

The atom style defined does not have this attribute.

Fix heat group has no atoms

Self−explanatory.

Fix heat kinetic energy went negative

This will cause the velocity rescaling about to be performed by fix heat to be invalid.

Fix in variable not computed at compatible time

Fixes generate their values on specific timesteps. The variable is requesting the values on a non−allowed timestep.

Fix langevin period must be > 0.0

The time window for temperature relaxation must be > 0

Fix momentum group has no atoms

Self−explanatory.

Fix move cannot define z or vz variable for 2d problem

Self−explanatory.

Fix move cannot have 0 length rotation vector

Self−explanatory.

Fix move cannot rotate aroung non z−axis for 2d problem

Self−explanatory.

Fix move cannot set linear z motion for 2d problem

Self−explanatory.

Fix move cannot set wiggle z motion for 2d problem

Self−explanatory.

Fix msst compute ID does not compute potential energy

Self−explanatory.

Fix msst compute ID does not compute pressure

Self−explanatory.

Fix msst compute ID does not compute temperature

Self−explanatory.

Fix msst requires a periodic box

Self−explanatory.

Fix msst tscale must satisfy 0 <= tscale < 1

Self−explanatory.

Fix npt/nph has tilted box too far − box flips are not yet implemented

This feature has not yet been added. However, if you are applying an off−diagonal pressure to a fluid, the box may want to tilt indefinitely, because the fluid cannot support the pressure you are imposing.

Fix nve/asphere cannot be used with atom attributes diameter or rmass

These attributes override the shape and mass settings, so cannot be used.

Fix nve/asphere requires atom attributes angmom, quat, torque, shape

An atom style that specifies these quantities is needed.

Fix nve/asphere requires extended particles

This fix can only be used for particles with a shape setting.

Fix nve/sphere requires atom attribute diameter or shape

An atom style that specifies these quantities is needed.

Fix nve/sphere requires atom attribute mu

An atom style with this attribute is needed.

Fix nve/sphere requires atom attributes omega, torque

An atom style with these attributes is needed.

Fix nve/sphere requires extended particles

This fix can only be used for particles of a finite size.

Fix nve/sphere requires spherical particle shapes

Self−explanatory.

Fix nvt/nph/npt asphere cannot be used with atom attributes diameter or rmass

Those attributes are for spherical particles.

Fix nvt/nph/npt asphere requires atom attributes quat, angmom, torque, shape

Those attributes are what are used to define aspherical particles.

Fix nvt/nph/npt asphere requires extended particles

The shape setting for a particle in the fix group has shape = 0.0, which means it is a point particle.

Fix nvt/nph/npt sphere requires atom attribute diameter or shape

An atom style that specifies these quantities is needed.

Fix nvt/nph/npt sphere requires atom attributes omega, torque

Those attributes are what are used to define spherical particles.

Fix nvt/npt/nph damping parameters must be > 0.0

Self−explanatory.

Fix nvt/sphere requires extended particles

This fix can only be used for particles of a finite size.

Fix nvt/sphere requires spherical particle shapes

Self−explanatory.

Fix orient/fcc file open failed

The fix orient/fcc command could not open a specified file.

Fix orient/fcc file read failed

The fix orient/fcc command could not read the needed parameters from a specified file.

Fix orient/fcc found self twice

The neighbor lists used by fix orient/fcc are messed up. If this error occurs, it is likely a bug, so send an email to the developers.

Fix peri neigh does not exist

Somehow a fix that the pair style defines has been deleted.

Fix pour region ID does not exist

Self−explanatory.

Fix pour region cannot be dynamic

Only static regions can be used with fix pour.

Fix pour region does not support a bounding box

Not all regions represent bounded volumes. You cannot use such a region with the fix pour command.

Fix pour requires atom attributes radius, rmass

The atom style defined does not have these attributes.

Fix press/berendsen damping parameters must be > 0.0

Self−explanatory.

Fix qeq/comb group has no atoms

Self−explanatory.

Fix qeq/comb requires atom attribute q

An atom style with charge must be used to perform charge equilibration.

Fix reax/bonds numbonds > nsbmax_most

The limit of the number of bonds expected by the ReaxFF force field was exceeded.

Fix recenter group has no atoms

Self−explanatory.

Fix rigid atom has non−zero image flag in a non−periodic dimension

You cannot set image flags for non−periodic dimensions.

Fix rigid molecule requires atom attribute molecule

Self−explanatory.

Fix rigid/nvt period must be > 0.0

Self−explanatory

Fix rigid: Bad principal moments

The principal moments of inertia computed for a rigid body are not within the required tolerances.

Fix shake cannot be used with minimization
: Cannot use fix shake while doing an energy minimization since it turns off bonds that should contribute to the energy.

Fix spring couple group ID does not exist
: Self−explanatory.

Fix srd lamda must be >= 0.6 of SRD grid size
: This is a requirement for accuracy reasons.

Fix srd requires SRD particles all have same mass
: Self−explanatory.

Fix srd requires ghost atoms store velocity
: Use the communicate vel yes command to enable this.

Fix srd requires newton pair on
: Self−explanatory.

Fix store/state compute array is accessed out−of−range
: Self−explanatory.

Fix store/state compute does not calculate a per−atom array
: The compute calculates a per−atom vector.

Fix store/state compute does not calculate a per−atom vector
: The compute calculates a per−atom vector.

Fix store/state compute does not calculate per−atom values
: Computes that calculate global or local quantities cannot be used with fix store/state.

Fix store/state fix array is accessed out−of−range
: Self−explanatory.

Fix store/state fix does not calculate a per−atom array
: The fix calculates a per−atom vector.

Fix store/state fix does not calculate a per−atom vector
: The fix calculates a per−atom array.

Fix store/state fix does not calculate per−atom values
: Fixes that calculate global or local quantities cannot be used with fix store/state.

Fix store/state for atom property that isn't allocated
: Self−explanatory.

Fix store/state variable is not atom−style variable
: Only atom−style variables calculate per−atom quantities.

Fix temp/berendsen period must be > 0.0
: Self−explanatory.

Fix thermal/conductivity swap value must be positive
: Self−explanatory.

Fix tmd must come after integration fixes
: Any fix tmd command must appear in the input script after all time integration fixes (nve, nvt, npt). See the fix tmd documentation for details.

Fix ttm electron temperatures must be > 0.0
: Self−explanatory.

Fix ttm electronic_density must be > 0.0
: Self−explanatory.

Fix ttm electronic_specific_heat must be > 0.0
: Self−explanatory.

Fix ttm electronic_thermal_conductivity must be >= 0.0
: Self−explanatory.

Fix ttm gamma_p must be > 0.0
: Self−explanatory.

Fix ttm gamma_s must be >= 0.0
: Self−explanatory.

Fix ttm number of nodes must be > 0
Self−explanatory.
Fix ttm v_0 must be >= 0.0
Self−explanatory.
Fix used in compute atom/molecule not computed at compatible time
The fix must produce per−atom quantities on timesteps that the compute needs them.
Fix used in compute reduce not computed at compatible time
Fixes generate their values on specific timesteps. Compute sum is requesting a value on a non−allowed timestep.
Fix viscosity swap value must be positive
Self−explanatory.
Fix viscosity vtarget value must be positive
Self−explanatory.
Fix wall cutoff <= 0.0
Self−explanatory.
Fix wall/colloid cannot be used with atom attribute diameter
Only finite−size particles defined by the shape command can be used.
Fix wall/colloid requires atom attribute shape
Self−explanatory.
Fix wall/colloid requires extended particles
Self−explanatory.
Fix wall/colloid requires spherical particles
Self−explanatory.
Fix wall/gran is incompatible with Pair style
Must use a granular pair style to define the parameters needed for this fix.
Fix wall/gran requires atom attributes radius, omega, torque
The atom style defined does not have these attributes.
Fix wall/region colloid cannot be used with atom attribute diameter
Only finite−size particles defined by the shape command can be used.
Fix wall/region colloid requires atom attribute shape
Self−explanatory.
Fix wall/region colloid requires extended particles
Self−explanatory.
Fix wall/region colloid requires spherical particles
Self−explanatory.
Fix wall/region cutoff <= 0.0
Self−explanatory.
Fix_modify order must be 3 or 5
Self−explanatory.
Fix_modify pressure ID does not compute pressure
The compute ID assigned to the fix must compute pressure.
Fix_modify temperature ID does not compute temperature
The compute ID assigned to the fix must compute temperature.
Found no restart file matching pattern
When using a "*" in the restart file name, no matching file was found.
GPU is not the first fix for this run
This is the way the fix must be defined in your input script.
GPU library not compiled for this accelerator
The GPU library was not built for your accelerator. Check the arch flag in lib/gpu.
Gmask function in equal−style variable formula
Gmask is per−atom operation.
Gravity changed since fix pour was created

Gravity must be static and not dynamic for use with fix pour.

Gravity must point in −y to use with fix pour in 2d

Gravity must be pointing "down" in a 2d box.

Gravity must point in −z to use with fix pour in 3d

Gravity must be pointing "down" in a 3d box, i.e. theta = 180.0.

Grmask function in equal−style variable formula

Grmask is per−atom operation.

Group ID does not exist

A group ID used in the group command does not exist.

Group ID in variable formula does not exist

Self−explanatory.

Group command before simulation box is defined

The group command cannot be used before a read_data, read_restart, or create_box command.

Group region ID does not exist

A region ID used in the group command does not exist.

Illegal ... command

Self−explanatory. Check the input script syntax and compare to the documentation for the command. You can use −echo screen as a command−line option when running DSMC to see the offending line.

Illegal COMB parameter

One or more of the coefficients defined in the potential file is invalid.

Illegal Stillinger−Weber parameter

One or more of the coefficients defined in the potential file is invalid.

Illegal Tersoff parameter

One or more of the coefficients defined in the potential file is invalid.

Illegal chemical element names

The name is too long to be a chemical element.

Illegal fix gpu command

Self−explanatory.

Illegal number of angle table entries

There must be at least 2 table entries.

Illegal number of bond table entries

There must be at least 2 table entries.

Illegal number of pair table entries

There must be at least 2 table entries.

Illegal simulation box

The lower bound of the simulation box is greater than the upper bound.

Improper atom missing in delete_bonds

The delete_bonds command cannot find one or more atoms in a particular improper on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid improper.

Improper atom missing in set command

The set command cannot find one or more atoms in a particular improper on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid improper.

Improper atoms %d %d %d %d missing on proc %d at step

One or more of 4 atoms needed to compute a particular improper are missing on this processor. Typically this is because the pairwise cutoff is set too short or the improper has blown apart and an atom is too far away.

Improper coeff for hybrid has invalid style

Improper style hybrid uses another improper style as one of its coefficients. The improper style used in the improper_coeff command or read from a restart file is not recognized.

Improper coeffs are not set

No improper coefficients have been assigned in the data file or via the improper_coeff command.

Improper style hybrid cannot have hybrid as an argument

Self−explanatory.

Improper style hybrid cannot have none as an argument

Self−explanatory.

Improper style hybrid cannot use same improper style twice

Self−explanatory.

Improper_coeff command before improper_style is defined

Coefficients cannot be set in the data file or via the improper_coeff command until an improper_style has been assigned.

Improper_coeff command before simulation box is defined

The improper_coeff command cannot be used before a read_data, read_restart, or create_box command.

Improper_coeff command when no impropers allowed

The chosen atom style does not allow for impropers to be defined.

Improper_style command when no impropers allowed

The chosen atom style does not allow for impropers to be defined.

Impropers assigned incorrectly

Impropers read in from the data file were not assigned correctly to atoms. This means there is something invalid about the topology definitions.

Impropers defined but no improper types

The data file header lists improper but no improper types.

Inconsistent iparam/jparam values in fix bond/create command

If itype and jtype are the same, then their maxbond and newtype settings must also be the same.

Incorrect args for angle coefficients

Self−explanatory. Check the input script or data file.

Incorrect args for bond coefficients

Self−explanatory. Check the input script or data file.

Incorrect args for dihedral coefficients

Self−explanatory. Check the input script or data file.

Incorrect args for improper coefficients

Self−explanatory. Check the input script or data file.

Incorrect args for pair coefficients

Self−explanatory. Check the input script or data file.

Incorrect args in pair_style command

Self−explanatory.

Incorrect atom format in data file

Number of values per atom line in the data file is not consistent with the atom style.

Incorrect boundaries with slab Ewald

Must have periodic x,y dimensions and non−periodic z dimension to use 2d slab option with Ewald.

Incorrect boundaries with slab PPPM

Must have periodic x,y dimensions and non−periodic z dimension to use 2d slab option with PPPM.

Incorrect element names in EAM potential file

The element names in the EAM file do not match those requested.

Incorrect format in COMB potential file

Incorrect number of words per line in the potential file.

Incorrect format in MEAM potential file

Incorrect number of words per line in the potential file.

Incorrect format in NEB coordinate file

Self−explanatory.

Incorrect format in Stillinger−Weber potential file

Incorrect number of words per line in the potential file.

Incorrect format in TMD target file

Format of file read by fix tmd command is incorrect.

Incorrect format in Tersoff potential file

Incorrect number of words per line in the potential file.

Incorrect multiplicity arg for dihedral coefficients

Self−explanatory. Check the input script or data file.

Incorrect sign arg for dihedral coefficients

Self−explanatory. Check the input script or data file.

Incorrect velocity format in data file

Each atom style defines a format for the Velocity section of the data file. The read−in lines do not match.

Incorrect weight arg for dihedral coefficients

Self−explanatory. Check the input script or data file.

Index between variable brackets must be positive

Self−explanatory.

Indexed per−atom vector in variable formula without atom map

Accessing a value from an atom vector requires the ability to lookup an atom index, which is provided by an atom map. An atom map does not exist (by default) for non−molecular problems. Using the atom_modify map command will force an atom map to be created.

Induced tilt by displace_box is too large

The final tilt value must be between −1/2 and 1/2 of the perpendicular box length.

Initial temperatures not all set in fix ttm

Self−explantory.

Input line too long after variable substitution

This is a hard (very large) limit defined in the input.cpp file.

Input line too long: %s

This is a hard (very large) limit defined in the input.cpp file.

Insertion region extends outside simulation box

Region specified with fix pour command extends outside the global simulation box.

Insufficient Jacobi rotations for POEMS body

Eigensolve for rigid body was not sufficiently accurate.

Insufficient Jacobi rotations for rigid body

Eigensolve for rigid body was not sufficiently accurate.

Insufficient memory on accelerator.

Self−explanatory.

Invalid Boolean syntax in if command

Self−explanatory.

Invalid REAX atom type

There is a mis−match between DSMC atom types and the elements listed in the ReaxFF force field file.

Invalid angle style

The choice of angle style is unknown.

Invalid angle table length

Length must be 2 or greater.

Invalid angle type in Angles section of data file

Angle type must be positive integer and within range of specified angle types.

Invalid angle type index for fix shake

Self−explanatory.

Invalid atom ID in Angles section of data file

Atom IDs must be positive integers and within range of defined atoms.

Invalid atom ID in Atoms section of data file

Atom IDs must be positive integers.

Invalid atom ID in Bonds section of data file

Atom IDs must be positive integers and within range of defined atoms.

Invalid atom ID in Dihedrals section of data file

Atom IDs must be positive integers and within range of defined atoms.

Invalid atom ID in Impropers section of data file

Atom IDs must be positive integers and within range of defined atoms.

Invalid atom ID in Velocities section of data file

Atom IDs must be positive integers and within range of defined atoms.

Invalid atom mass for fix shake

Mass specified in fix shake command must be > 0.0.

Invalid atom style

The choice of atom style is unknown.

Invalid atom type in Atoms section of data file

Atom types must range from 1 to specified # of types.

Invalid atom type in create_atoms command

The create_box command specified the range of valid atom types. An invalid type is being requested.

Invalid atom type in fix bond/create command

Self−explanatory.

Invalid atom type in neighbor exclusion list

Atom types must range from 1 to Ntypes inclusive.

Invalid atom type index for fix shake

Atom types must range from 1 to Ntypes inclusive.

Invalid atom types in pair_write command

Atom types must range from 1 to Ntypes inclusive.

Invalid atom vector in variable formula

The atom vector is not recognized.

Invalid attribute in dump custom command

Self−explantory.

Invalid attribute in dump local command

Self−explantory.

Invalid attribute in dump modify command

Self−explantory.

Invalid bond style

The choice of bond style is unknown.

Invalid bond table length

Length must be 2 or greater.

Invalid bond type in Bonds section of data file

Bond type must be positive integer and within range of specified bond types.

Invalid bond type in fix bond/break command

Self−explanatory.

Invalid bond type in fix bond/create command

Self−explanatory.

Invalid bond type index for fix shake

Self−explanatory. Check the fix shake command in the input script.

Invalid coeffs for this dihedral style

Cannot set class 2 coeffs in data file for this dihedral style.

Invalid command−line argument

One or more command−line arguments is invalid. Check the syntax of the command you are using to launch DSMC.

Invalid compute ID in variable formula

The compute is not recognized.

Invalid compute style

Self−explanatory.

Invalid cutoff in communicate command

Specified cutoff must be >= 0.0.

Invalid cutoffs in pair_write command

Inner cutoff must be larger than 0.0 and less than outer cutoff.

Invalid d1 or d2 value for pair colloid coeff
Neither d1 or d2 can be < 0.
Invalid data file section: Angle Coeffs
Atom style does not allow angles.
Invalid data file section: AngleAngle Coeffs
Atom style does not allow impropers.
Invalid data file section: AngleAngleTorsion Coeffs
Atom style does not allow dihedrals.
Invalid data file section: AngleTorsion Coeffs
Atom style does not allow dihedrals.
Invalid data file section: Angles
Atom style does not allow angles.
Invalid data file section: Bond Coeffs
Atom style does not allow bonds.
Invalid data file section: BondAngle Coeffs
Atom style does not allow angles.
Invalid data file section: BondBond Coeffs
Atom style does not allow angles.
Invalid data file section: BondBond13 Coeffs
Atom style does not allow dihedrals.
Invalid data file section: Bonds
Atom style does not allow bonds.
Invalid data file section: Dihedral Coeffs
Atom style does not allow dihedrals.
Invalid data file section: Dihedrals
Atom style does not allow dihedrals.
Invalid data file section: EndBondTorsion Coeffs
Atom style does not allow dihedrals.
Invalid data file section: Improper Coeffs
Atom style does not allow impropers.
Invalid data file section: Impropers
Atom style does not allow impropers.
Invalid data file section: MiddleBondTorsion Coeffs
Atom style does not allow dihedrals.
Invalid delta_conf in tad command
The value must be between 0 and 1 inclusive.
Invalid density in Atoms section of data file
Density value cannot be <= 0.0.
Invalid dihedral style
The choice of dihedral style is unknown.
Invalid dihedral type in Dihedrals section of data file
Dihedral type must be positive integer and within range of specified dihedral types.
Invalid dipole line in data file
Self−explanatory.
Invalid dipole value
Self−explanatory.
Invalid dump dcd filename
Filenames used with the dump dcd style cannot be binary or compressed or cause multiple files to be written.
Invalid dump frequency
Dump frequency must be 1 or greater.
Invalid dump style

The choice of dump style is unknown.

Invalid dump xtc filename

Filenames used with the dump xtc style cannot be binary or compressed or cause multiple files to be written.

Invalid dump xyz filename

Filenames used with the dump xyz style cannot be binary or cause files to be written by each processor.

Invalid dump_modify threshhold operator

Operator keyword used for threshold specification in not recognized.

Invalid fix ID in variable formula

The fix is not recognized.

Invalid fix ave/time off column

Self−explantory.

Invalid fix box/relax command for a 2d simulation

Fix box/relax styles involving the z dimension cannot be used in a 2d simulation.

Invalid fix box/relax command pressure settings

If multiple dimensions are coupled, those dimensions must be specified.

Invalid fix box/relax pressure settings

Settings for coupled dimensions must be the same.

Invalid fix nvt/npt/nph command for a 2d simulation

Cannot control z dimension in a 2d model.

Invalid fix nvt/npt/nph command pressure settings

If multiple dimensions are coupled, those dimensions must be specified.

Invalid fix nvt/npt/nph pressure settings

Settings for coupled dimensions must be the same.

Invalid fix press/berendsen for a 2d simulation

The z component of pressure cannot be controlled for a 2d model.

Invalid fix press/berendsen pressure settings

Settings for coupled dimensions must be the same.

Invalid fix style

The choice of fix style is unknown.

Invalid flag in force field section of restart file

Unrecognized entry in restart file.

Invalid flag in header section of restart file

Unrecognized entry in restart file.

Invalid flag in type arrays section of restart file

Unrecognized entry in restart file.

Invalid frequency in temper command

Nevery must be > 0.

Invalid group ID in neigh_modify command

A group ID used in the neigh_modify command does not exist.

Invalid group function in variable formula

Group function is not recognized.

Invalid group in communicate command

Self−explanatory.

Invalid improper style

The choice of improper style is unknown.

Invalid improper type in Impropers section of data file

Improper type must be positive integer and within range of specified improper types.

Invalid keyword in angle table parameters

Self−explanatory.

Invalid keyword in bond table parameters

Self−explanatory.

Invalid keyword in compute angle/local command
Self−explanatory.
Invalid keyword in compute bond/local command
Self−explanatory.
Invalid keyword in compute dihedral/local command
Self−explanatory.
Invalid keyword in compute improper/local command
Self−explanatory.
Invalid keyword in compute pair/local command
Self−explanatory.
Invalid keyword in compute property/atom command
Self−explanatory.
Invalid keyword in compute property/local command
Self−explanatory.
Invalid keyword in compute property/molecule command
Self−explanatory.
Invalid keyword in dump cfg command
Self−explanatory.
Invalid keyword in pair table parameters
Keyword used in list of table parameters is not recognized.
Invalid keyword in thermo_style custom command
One or more specified keywords are not recognized.
Invalid kspace style
The choice of kspace style is unknown.
Invalid mass line in data file
Self−explanatory.
Invalid mass value
Self−explanatory.
Invalid math function in variable formula
Self−explanatory.
Invalid math/group/special function in variable formula
Self−explanatory.
Invalid option in lattice command for non−custom style
Certain lattice keywords are not supported unless the lattice style is "custom".
Invalid order of forces within respa levels
For respa, ordering of force computations within respa levels must obey certain rules. E.g. bonds cannot be compute less frequently than angles, pairwise forces cannot be computed less frequently than kspace, etc.
Invalid pair style
The choice of pair style is unknown.
Invalid pair table cutoff
Cutoffs in pair_coeff command are not valid with read−in pair table.
Invalid pair table length
Length of read−in pair table is invalid
Invalid radius in Atoms section of data file
Radius must be >= 0.0.
Invalid random number seed in fix ttm command
Random number seed must be > 0.
Invalid random number seed in set command
Random number seed must be > 0.
Invalid region style
The choice of region style is unknown.

Invalid replace values in compute reduce
: Self−explanatory.

Invalid run command N value
: The number of timesteps must fit in a 32−bit integer. If you want to run for more steps than this, perform multiple shorter runs.

Invalid run command start/stop value
: Self−explanatory.

Invalid run command upto value
: Self−explanatory.

Invalid seed for Marsaglia random # generator
: The initial seed for this random number generator must be a positive integer less than or equal to 900 million.

Invalid seed for Park random # generator
: The initial seed for this random number generator must be a positive integer.

Invalid shape line in data file
: Self−explanatory.

Invalid shape line in data file
: Self−explanatory.

Invalid shape value
: Self−explanatory.

Invalid shear direction for fix wall/gran
: Self−explanatory.

Invalid special function in variable formula
: Self−explanatory.

Invalid style in pair_write command
: Self−explanatory. Check the input script.

Invalid syntax in variable formula
: Self−explanatory.

Invalid t_event in prd command
: Self−explanatory.

Invalid t_event in tad command
: The value must be greater than 0.

Invalid thermo keyword in variable formula
: The keyword is not recognized.

Invalid tmax in tad command
: The value must be greater than 0.0.

Invalid type for dipole set
: Dipole command must set a type from 1−N where N is the number of atom types.

Invalid type for mass set
: Mass command must set a type from 1−N where N is the number of atom types.

Invalid type for shape set
: Atom type is out of bounds.

Invalid value in set command
: The value specified for the setting is invalid, likely because it is too small or too large.

Invalid variable evaluation in variable formula
: A variable used in a formula could not be evaluated.

Invalid variable in next command
: Self−explanatory.

Invalid variable name in variable formula
: Variable name is not recognized.

Invalid variable name
: Variable name used in an input script line is invalid.

Invalid variable style with next command
Variable styles *equal* and *world* cannot be used in a next command.
Invalid wiggle direction for fix wall/gran
Self−explanatory.
Invoked angle equil angle on angle style none
Self−explanatory.
Invoked angle single on angle style none
Self−explanatory.
Invoked bond equil distance on bond style none
Self−explanatory.
Invoked bond single on bond style none
Self−explanatory.
Invoked pair single on pair style none
A command (e.g. a dump) attempted to invoke the single() function on a pair style none, which is illegal. You are probably attempting to compute per−atom quantities with an undefined pair style.
KSpace style has not yet been set
Cannot use kspace_modify command until a kspace style is set.
KSpace style is incompatible with Pair style
Setting a kspace style requires that a pair style with a long−range Coulombic component be selected.
Keyword %s in MEAM parameter file not recognized
Self−explanatory.
Kspace style requires atom attribute q
The atom style defined does not have these attributes.
Label wasn't found in input script
Self−explanatory.
Lattice orient vectors are not orthogonal
The three specified lattice orientation vectors must be mutually orthogonal.
Lattice orient vectors are not right−handed
The three specified lattice orientation vectors must create a right−handed coordinate system such that a1 cross a2 = a3.
Lattice primitive vectors are collinear
The specified lattice primitive vectors do not for a unit cell with non−zero volume.
Lattice settings are not compatible with 2d simulation
One or more of the specified lattice vectors has a non−zero z component.
Lattice spacings are invalid
Each x,y,z spacing must be > 0.
Lattice style incompatible with simulation dimension
2d simulation can use sq, sq2, or hex lattice. 3d simulation can use sc, bcc, or fcc lattice.
Log of zero/negative value in variable formula
Self−explanatory.
MEAM library error %d
A call to the MEAM Fortran library returned an error.
MPI_LMP_BIGINT and bigint in lmptype.h are not compatible
The size of the MPI datatype does not match the size of a bigint.
MPI_LMP_TAGINT and tagint in lmptype.h are not compatible
The size of the MPI datatype does not match the size of a tagint.
Mass command before simulation box is defined
The mass command cannot be used before a read_data, read_restart, or create_box command.
Min_style command before simulation box is defined
The min_style command cannot be used before a read_data, read_restart, or create_box command.
Minimization could not find thermo_pe compute
This compute is created by the thermo command. It must have been explicitly deleted by a uncompute

command.

Minimize command before simulation box is defined

The minimize command cannot be used before a read_data, read_restart, or create_box command.

Mismatched brackets in variable

Self−explanatory.

Mismatched compute in variable formula

A compute is referenced incorrectly or a compute that produces per−atom values is used in an equal−style variable formula.

Mismatched fix in variable formula

A fix is referenced incorrectly or a fix that produces per−atom values is used in an equal−style variable formula.

Mismatched variable in variable formula

A variable is referenced incorrectly or an atom−style variable that produces per−atom values is used in an equal−style variable formula.

Molecular data file has too many atoms

These kids of data files are currently limited to a number of atoms that fits in a 32−bit integer.

Molecule count changed in compute atom/molecule

Number of molecules must remain constant over time.

Molecule count changed in compute com/molecule

Number of molecules must remain constant over time.

Molecule count changed in compute gyration/molecule

Number of molecules must remain constant over time.

Molecule count changed in compute msd/molecule

Number of molecules must remain constant over time.

Molecule count changed in compute property/molecule

Number of molecules must remain constant over time.

More than one fix deform

Only one fix deform can be defined at a time.

More than one fix freeze

Only one of these fixes can be defined, since the granular pair potentials access it.

More than one fix shake

Only one fix shake can be defined.

Must define angle_style before Angle Coeffs

Must use an angle_style command before reading a data file that defines Angle Coeffs.

Must define angle_style before BondAngle Coeffs

Must use an angle_style command before reading a data file that defines Angle Coeffs.

Must define angle_style before BondBond Coeffs

Must use an angle_style command before reading a data file that defines Angle Coeffs.

Must define bond_style before Bond Coeffs

Must use a bond_style command before reading a data file that defines Bond Coeffs.

Must define dihedral_style before AngleAngleTorsion Coeffs

Must use a dihedral_style command before reading a data file that defines AngleAngleTorsion Coeffs.

Must define dihedral_style before AngleTorsion Coeffs

Must use a dihedral_style command before reading a data file that defines AngleTorsion Coeffs.

Must define dihedral_style before BondBond13 Coeffs

Must use a dihedral_style command before reading a data file that defines BondBond13 Coeffs.

Must define dihedral_style before Dihedral Coeffs

Must use a dihedral_style command before reading a data file that defines Dihedral Coeffs.

Must define dihedral_style before EndBondTorsion Coeffs

Must use a dihedral_style command before reading a data file that defines EndBondTorsion Coeffs.

Must define dihedral_style before MiddleBondTorsion Coeffs

Must use a dihedral_style command before reading a data file that defines MiddleBondTorsion Coeffs.

Must define improper_style before AngleAngle Coeffs
: Must use an improper_style command before reading a data file that defines AngleAngle Coeffs.

Must define improper_style before Improper Coeffs
: Must use an improper_style command before reading a data file that defines Improper Coeffs.

Must define pair_style before Pair Coeffs
: Must use a pair_style command before reading a data file that defines Pair Coeffs.

Must have more than one processor partition to temper
: Cannot use the temper command with only one processor partition. Use the −partition command−line option.

Must read Atoms before Angles
: The Atoms section of a data file must come before an Angles section.

Must read Atoms before Bonds
: The Atoms section of a data file must come before a Bonds section.

Must read Atoms before Dihedrals
: The Atoms section of a data file must come before a Dihedrals section.

Must read Atoms before Impropers
: The Atoms section of a data file must come before an Impropers section.

Must read Atoms before Velocities
: The Atoms section of a data file must come before a Velocities section.

Must set both respa inner and outer
: Cannot use just the inner or outer option with respa without using the other.

Must specify a region in fix deposit
: The region keyword must be specified with this fix.

Must specify a region in fix pour
: The region keyword must be specified with this fix.

Must use −in switch with multiple partitions
: A multi−partition simulation cannot read the input script from stdin. The −in command−line option must be used to specify a file.

Must use a block or cylinder region with fix pour
: Self−explanatory.

Must use a block region with fix pour for 2d simulations
: Self−explanatory.

Must use a bond style with TIP4P potential
: TIP4P potentials assume bond lengths in water are constrained by a fix shake command.

Must use a molecular atom style with fix poems molecule
: Self−explanatory.

Must use a z−axis cylinder with fix pour
: The axis of the cylinder region used with the fix pour command must be oriented along the z dimension.

Must use an angle style with TIP4P potential
: TIP4P potentials assume angles in water are constrained by a fix shake command.

Must use atom style with molecule IDs with fix bond/swap
: Self−explanatory.

Must use pair_style comb with fix qeq/comb
: Self−explanatory.

Must use variable energy with fix addforce
: Must define an energy vartiable when applyting a dynamic force during minimization.

NEB command before simulation box is defined
: Self−explanatory.

NEB requires damped dynamics minimizer
: Use a different minimization style.

NEB requires use of fix neb
: Self−explanatory.

Needed topology not in data file
: The header of the data file indicated that bonds or angles or dihedrals or impropers would be included, but they were not present.

Neigh_modify exclude molecule requires atom attribute molecule
: Self−explanatory.

Neigh_modify include group != atom_modify first group
: Self−explanatory.

Neighbor delay must be 0 or multiple of every setting
: The delay and every parameters set via the neigh_modify command are inconsistent. If the delay setting is non−zero, then it must be a multiple of the every setting.

Neighbor include group not allowed with ghost neighbors
: This is a current restriction within DSMC.

Neighbor list overflow, boost neigh_modify one or page
: There are too many neighbors of a single atom. Use the neigh_modify command to increase the neighbor page size and the max number of neighbors allowed for one atom.

Neighbor multi not yet enabled for ghost neighbors
: This is a current restriction within DSMC.

Neighbor multi not yet enabled for granular
: Self−explanatory.

Neighbor multi not yet enabled for rRESPA
: Self−explanatory.

Neighbor page size must be >= 10x the one atom setting
: This is required to prevent wasting too much memory.

Neighbors of ghost atoms only allowed for full neighbor lists
: This is a current restriction within DSMC.

New bond exceeded bonds per atom in fix bond/create
: See the read_data command for info on setting the "extra bond per atom" header value to allow for additional bonds to be formed.

New bond exceeded special list size in fix bond/create
: See the special_bonds extra command for info on how to leave space in the special bonds list to allow for additional bonds to be formed.

Newton bond change after simulation box is defined
: The newton command cannot be used to change the newton bond value after a read_data, read_restart, or create_box command.

No angle style is defined for compute angle/local
: Self−explanatory.

No angles allowed with this atom style
: Self−explanatory. Check data file.

No atoms in data file
: The header of the data file indicated that atoms would be included, but they were not present.

No basis atoms in lattice
: Basis atoms must be defined for lattice style user.

No bond style is defined for compute bond/local
: Self−explanatory.

No bonds allowed with this atom style
: Self−explanatory. Check data file.

No dihedral style is defined for compute dihedral/local
: Self−explanatory.

No dihedrals allowed with this atom style
: Self−explanatory. Check data file.

No dump custom arguments specified
: The dump custom command requires that atom quantities be specified to output to dump file.

No dump local arguments specified
: Self−explanatory.

No fix gravity defined for fix pour
: Cannot add poured particles without gravity to move them.

No improper style is defined for compute improper/local
: Self−explanatory.

No impropers allowed with this atom style
: Self−explanatory. Check data file.

No matching element in EAM potential file
: The EAM potential file does not contain elements that match the requested elements.

No pair hbond/dreiding coefficients set
: Self−explanatory.

No pair style defined for compute group/group
: Cannot calculate group interactions without a pair style defined.

No pair style is defined for compute pair/local
: Self−explanatory.

No pair style is defined for compute property/local
: Self−explanatory.

No rigid bodies defined
: The fix specification did not end up defining any rigid bodies.

Non digit character between brackets in variable
: Self−explantory.

Non integer # of swaps in temper command
: Swap frequency in temper command must evenly divide the total # of timesteps.

One or more atoms belong to multiple rigid bodies
: Two or more rigid bodies defined by the fix rigid command cannot contain the same atom.

One or zero atoms in rigid body
: Any rigid body defined by the fix rigid command must contain 2 or more atoms.

Out of range atoms − cannot compute PPPM
: One or more atoms are attempting to map their charge to a PPPM grid point that is not owned by a processor. This is likely for one of two reasons, both of them bad. First, it may mean that an atom near the boundary of a processor's sub−domain has moved more than 1/2 the neighbor skin distance without neighbor lists being rebuilt and atoms being migrated to new processors. This also means you may be missing pairwise interactions that need to be computed. The solution is to change the re−neighboring criteria via the neigh_modify command. The safest settings are "delay 0 every 1 check yes". Second, it may mean that an atom has moved far outside a processor's sub−domain or even the entire simulation box. This indicates bad physics, e.g. due to highly overlapping atoms, too large a timestep, etc.

Overlapping large/large in pair colloid
: This potential is infinite when there is an overlap.

Overlapping small/large in pair colloid
: This potential is inifinte when there is an overlap.

POEMS fix must come before NPT/NPH fix
: NPT/NPH fix must be defined in input script after all poems fixes, else the fix contribution to the pressure virial is incorrect.

PPPM grid is too large
: The global PPPM grid is larger than OFFSET in one or more dimensions. OFFSET is currently set to 4096. You likely need to decrease the requested precision.

PPPM order cannot be greater than %d
: Self−explanatory.

PPPM order has been reduced to 0
: DSMC has attempted to reduce the PPPM order to enable the simulation to run, but can reduce the order no further. Try increasing the accuracy of PPPM by reducing the tolerance size, thus inducing a larger

PPPM grid.

PRD command before simulation box is defined

The prd command cannot be used before a read_data, read_restart, or create_box command.

PRD nsteps must be multiple of t_event

Self−explanatory.

PRD t_corr must be multiple of t_event

Self−explanatory.

Pair coeff for hybrid has invalid style

Style in pair coeff must have been listed in pair_style command.

Pair cutoff < Respa interior cutoff

One or more pairwise cutoffs are too short to use with the specified rRESPA cutoffs.

Pair dipole/cut requires atom attributes q, mu, torque, dipole

An atom style that specifies these quantities is needed.

Pair distance < table inner cutoff

Two atoms are closer together than the pairwise table allows.

Pair distance > table outer cutoff

Two atoms are further apart than the pairwise table allows.

Pair dpd requires ghost atoms store velocity

Use the communicate vel yes command to enable this.

Pair gayberne cannot be used with atom attribute diameter

Finite−size particles must be defined with the shape command.

Pair gayberne epsilon a,b,c coeffs are not all set

Each atom type involved in pair_style gayberne must have these 3 coefficients set at least once.

Pair gayberne requires atom attributes quat, torque, shape

An atom style that defines these attributes must be used.

Pair granular requires atom attributes radius, omega, torque

The atom style defined does not have these attributes.

Pair granular requires ghost atoms store velocity

Use the communicate vel yes command to enable this.

Pair granular with shear history requires newton pair off

This is a current restriction of the implementation of pair granular styles with history.

Pair hybrid sub−style does not support single call

You are attempting to invoke a single() call on a pair style that doesn't support it.

Pair hybrid sub−style is not used

No pair_coeff command used a sub−style specified in the pair_style command.

Pair inner cutoff < Respa interior cutoff

One or more pairwise cutoffs are too short to use with the specified rRESPA cutoffs.

Pair inner cutoff >= Pair outer cutoff

The specified cutoffs for the pair style are inconsistent.

Pair lubricate cannot be used with atom attributes diameter or rmass

These attributes override the shape and mass settings, so cannot be used.

Pair lubricate requires atom attribute omega or angmom

An atom style that defines these attributes must be used.

Pair lubricate requires atom attributes torque and shape

An atom style that defines these attributes must be used.

Pair lubricate requires extended particles

This pair style can only be used for particles with a shape setting.

Pair lubricate requires ghost atoms store velocity

Use the communicate vel yes command to enable this.

Pair lubricate requires spherical, mono−disperse particles

This is a current restriction of this pair style.

Pair peri lattice is not identical in x, y, and z

The lattice defined by the lattice command must be cubic.

Pair peri requires a lattice be defined

Use the lattice command for this purpose.

Pair peri requires an atom map, see atom_modify

Even for atomic systems, an atom map is required to find Peridynamic bonds. Use the atom_modify command to define one.

Pair resquared cannot be used with atom attribute diameter

This attribute overrides the shape settings, so cannot be used.

Pair resquared epsilon a,b,c coeffs are not all set

Self−explanatory.

Pair resquared epsilon and sigma coeffs are not all set

Self−explanatory.

Pair resquared requires atom attributes quat, torque, shape

An atom style that defines these attributes must be used.

Pair style AIREBO requires atom IDs

This is a requirement to use the AIREBO potential.

Pair style AIREBO requires newton pair on

See the newton command. This is a restriction to use the AIREBO potential.

Pair style COMB requires atom IDs

This is a requirement to use the AIREBO potential.

Pair style COMB requires atom attribute q

Self−explanatory.

Pair style COMB requires newton pair on

See the newton command. This is a restriction to use the COMB potential.

Pair style MEAM requires newton pair on

See the newton command. This is a restriction to use the MEAM potential.

Pair style Stillinger−Weber requires atom IDs

This is a requirement to use the SW potential.

Pair style Stillinger−Weber requires newton pair on

See the newton command. This is a restriction to use the SW potential.

Pair style Tersoff requires atom IDs

This is a requirement to use the Tersoff potential.

Pair style Tersoff requires newton pair on

See the newton command. This is a restriction to use the Tersoff potential.

Pair style born/coul/long requires atom attribute q

An atom style that defines this attribute must be used.

Pair style buck/coul/cut requires atom attribute q

The atom style defined does not have this attribute.

Pair style buck/coul/long requires atom attribute q

The atom style defined does not have these attributes.

Pair style coul/cut requires atom attribute q

The atom style defined does not have these attributes.

Pair style does not support bond_style quartic

The pair style does not have a single() function, so it can not be invoked by bond_style quartic.

Pair style does not support compute group/group

The pair_style does not have a single() function, so it cannot be invokded by the compute group/group command.

Pair style does not support compute pair/local

The pair style does not have a single() function, so it can not be invoked by fix bond/swap.

Pair style does not support compute property/local

The pair style does not have a single() function, so it can not be invoked by fix bond/swap.

Pair style does not support fix bond/swap

The pair style does not have a single() function, so it can not be invoked by fix bond/swap.

Pair style does not support pair_write

The pair style does not have a single() function, so it can not be invoked by pair write.

Pair style does not support rRESPA inner/middle/outer

You are attempting to use rRESPA options with a pair style that does not support them.

Pair style granular with history requires atoms have IDs

Atoms in the simulation do not have IDs, so history effects cannot be tracked by the granular pair potential.

Pair style hbond/dreiding requires an atom map, see atom_modify

Self−explanatory.

Pair style hbond/dreiding requires atom IDs

Self−explanatory.

Pair style hbond/dreiding requires molecular system

Self−explanatory.

Pair style hbond/dreiding requires newton pair on

See the newton command for details.

Pair style hybrid cannot have hybrid as an argument

Self−explanatory.

Pair style hybrid cannot have none as an argument

Self−explanatory.

Pair style hybrid cannot use same pair style twice

The sub−style arguments of pair_style hybrid cannot be duplicated. Check the input script.

Pair style is incompatible with KSpace style

If a pair style with a long−range Coulombic component is selected, then a kspace style must also be used.

Pair style lj/charmm/coul/charmm requires atom attribute q

The atom style defined does not have these attributes.

Pair style lj/charmm/coul/long requires atom attribute q

The atom style defined does not have these attributes.

Pair style lj/class2/coul/cut requires atom attribute q

The atom style defined does not have this attribute.

Pair style lj/class2/coul/long requires atom attribute q

The atom style defined does not have this attribute.

Pair style lj/cut/coul/cut requires atom attribute q

The atom style defined does not have this attribute.

Pair style lj/cut/coul/long requires atom attribute q

The atom style defined does not have this attribute.

Pair style lj/cut/coul/long/tip4p requires atom IDs

There are no atom IDs defined in the system and the TIP4P potential requires them to find O,H atoms with a water molecule.

Pair style lj/cut/coul/long/tip4p requires atom attribute q

The atom style defined does not have these attributes.

Pair style lj/cut/coul/long/tip4p requires newton pair on

This is because the computation of constraint forces within a water molecule adds forces to atoms owned by other processors.

Pair style lj/gromacs/coul/gromacs requires atom attribute q

An atom_style with this attribute is needed.

Pair style peri_lps requires atom style peri

This is because atom style peri stores quantities needed by the peridynamic potential.

Pair style peri_pmb requires atom style peri

This is because atom style peri stores quantities needed by the peridynamic potential.

Pair style reax requires atom IDs

This is a requirement to use the ReaxFF potential.

Pair style reax requires newton pair on
: This is a requirement to use the ReaxFF potential.

Pair table cutoffs must all be equal to use with KSpace
: When using pair style table with a long−range KSpace solver, the cutoffs for all atom type pairs must all be the same, since the long−range solver starts at that cutoff.

Pair table parameters did not set N
: List of pair table parameters must include N setting.

Pair tersoff/zbl requires metal or real units
: This is a current restriction of this pair potential.

Pair yukawa/colloid cannot be used with atom attribute diameter
: Only finite−size particles defined by the shape command can be used.

Pair yukawa/colloid requires atom attribute shape
: Self−explanatory.

Pair yukawa/colloid requires spherical particles
: Self−explanatory.

Pair_coeff command before pair_style is defined
: Self−explanatory.

Pair_coeff command before simulation box is defined
: The pair_coeff command cannot be used before a read_data, read_restart, or create_box command.

Pair_modify command before pair_style is defined
: Self−explanatory.

Pair_write command before pair_style is defined
: Self−explanatory.

Particle on or inside fix wall surface
: Particles must be "exterior" to the wall in order for energy/force to be calculated.

Particle on or inside surface of region used in fix wall/region
: Particles must be "exterior" to the region surface in order for energy/force to be calculated.

Per−atom compute in equal−style variable formula
: Equal−style variables cannot use per−atom quantities.

Per−atom energy was not tallied on needed timestep
: You are using a thermo keyword that requires potentials to have tallied energy, but they didn't on this timestep. See the variable doc page for ideas on how to make this work.

Per−atom fix in equal−style variable formula
: Equal−style variables cannot use per−atom quantities.

Per−atom virial was not tallied on needed timestep
: You are using a thermo keyword that requires potentials to have tallied the virial, but they didn't on this timestep. See the variable doc page for ideas on how to make this work.

Per−processor system is too big
: The number of owned atoms plus ghost atoms on a single processor must fit in 32−bit integer.

Potential energy ID for fix neb does not exist
: Self−explanatory.

Potential file has duplicate entry
: The potential file for a SW or Tersoff potential has more than one entry for the same 3 ordered elements.

Potential file is missing an entry
: The potential file for a SW or Tersoff potential does not have a needed entry.

Power by 0 in variable formula
: Self−explanatory.

Pressure ID for fix box/relax does not exist
: The compute ID needed to compute pressure for the fix does not exist.

Pressure ID for fix modify does not exist
: Self−explanatory.

Pressure ID for fix npt/nph does not exist

Self−explanatory.

Pressure ID for fix press/berendsen does not exist

The compute ID needed to compute pressure for the fix does not exist.

Pressure ID for thermo does not exist

The compute ID needed to compute pressure for thermodynamics does not exist.

Pressure control can not be used with fix nvt/asphere

Self−explanatory.

Pressure control can not be used with fix nvt/sllod

Self−explanatory.

Pressure control can not be used with fix nvt/sphere

Self−explanatory.

Pressure control can not be used with fix nvt

Self−explanatory.

Pressure control must be used with fix nph/asphere

Self−explanatory.

Pressure control must be used with fix nph/sphere

Self−explanatory.

Pressure control must be used with fix nph

Self−explanatory.

Pressure control must be used with fix npt/asphere

Self−explanatory.

Pressure control must be used with fix npt/sphere

Self−explanatory.

Pressure control must be used with fix npt

Self−explanatory.

Processor count in z must be 1 for 2d simulation

Self−explanatory.

Processor partitions are inconsistent

The total number of processors in all partitions must match the number of processors DSMC is running on.

Processors command after simulation box is defined

The processors command cannot be used after a read_data, read_restart, or create_box command.

Quaternion creation numeric error

A numeric error occurred in the creation of a rigid body by the fix rigid command.

R0 < 0 for fix spring command

Equilibrium spring length is invalid.

Reax_defs.h setting for NATDEF is too small

Edit the setting in the ReaxFF library and re−compile the library and re−build DSMC.

Reax_defs.h setting for NNEIGHMAXDEF is too small

Edit the setting in the ReaxFF library and re−compile the library and re−build DSMC.

Region ID for compute reduce/region does not exist

Self−explanatory.

Region ID for compute temp/region does not exist

Self−explanatory.

Region ID for dump cfg does not exist

Self−explanatory.

Region ID for dump custom does not exist

Self−explanatory.

Region ID for fix addforce does not exist

Self−explanatory.

Region ID for fix ave/spatial does not exist

Self−explanatory.

Region ID for fix aveforce does not exist
Self−explanatory.
Region ID for fix deposit does not exist
Self−explanatory.
Region ID for fix evaporate does not exist
Self−explanatory.
Region ID for fix heat does not exist
Self−explanatory.
Region ID for fix setforce does not exist
Self−explanatory.
Region ID for fix wall/region does not exist
Self−explanatory.
Region ID in variable formula does not exist
Self−explanatory.
Region cannot have 0 length rotation vector
Self−explanatory.
Region intersect region ID does not exist
Self−explanatory.
Region union or intersect cannot be dynamic
The sub−regions can be dynamic, but not the combined region.
Region union region ID does not exist
One or more of the region IDs specified by the region union command does not exist.
Replacing a fix, but new style != old style
A fix ID can be used a 2nd time, but only if the style matches the previous fix. In this case it is assumed you with to reset a fix's parameters. This error may mean you are mistakenly re−using a fix ID when you do not intend to.
Replicate command before simulation box is defined
The replicate command cannot be used before a read_data, read_restart, or create_box command.
Replicate did not assign all atoms correctly
Atoms replicated by the replicate command were not assigned correctly to processors. This is likely due to some atom coordinates being outside a non−periodic simulation box.
Replicated molecular system atom IDs are too big
See the setting for the allowed atom ID size in the src/lmptype.h file.
Replicated system is too big
See the setting for bigint in the src/lmptype.h file.
Resetting timestep is not allowed with fix move
This is because fix move is moving atoms based on elapsed time.
Respa inner cutoffs are invalid
The first cutoff must be <= the second cutoff.
Respa levels must be >= 1
Self−explanatory.
Respa middle cutoffs are invalid
The first cutoff must be <= the second cutoff.
Reuse of compute ID
A compute ID cannot be used twice.
Reuse of dump ID
A dump ID cannot be used twice.
Reuse of region ID
A region ID cannot be used twice.
Rigid body has degenerate moment of inertia
Fix poems will only work with bodies (collections of atoms) that have non−zero principal moments of inertia. This means they must be 3 or more non−collinear atoms, even with joint atoms removed.

Rigid fix must come before NPT/NPH fix
: NPT/NPH fix must be defined in input script after all rigid fixes, else the rigid fix contribution to the pressure virial is incorrect.

Rmask function in equal−style variable formula
: Rmask is per−atom operation.

Run command before simulation box is defined
: The run command cannot be used before a read_data, read_restart, or create_box command.

Run command start value is after start of run
: Self−explanatory.

Run command stop value is before end of run
: Self−explanatory.

Run_style command before simulation box is defined
: The run_style command cannot be used before a read_data, read_restart, or create_box command.

SRD bin size for fix srd differs from user request
: Fix SRD had to adjust the bin size to fit the simulation box.

SRD bins for fix srd are not cubic enough
: The bin shape is not within tolerance of cubic.

Same dimension twice in fix ave/spatial
: Self−explanatory.

Set command before simulation box is defined
: The set command cannot be used before a read_data, read_restart, or create_box command.

Set command with no atoms existing
: No atoms are yet defined so the set command cannot be used.

Set region ID does not exist
: Region ID specified in set command does not exist.

Shake angles have different bond types
: All 3−atom angle−constrained SHAKE clusters specified by the fix shake command that are the same angle type, must also have the same bond types for the 2 bonds in the angle.

Shake atoms %d %d %d %d missing on proc %d at step
: The 4 atoms in a single shake cluster specified by the fix shake command are not all accessible to a processor. This probably means an atom has moved too far.

Shake atoms %d %d %d missing on proc %d at step
: The 3 atoms in a single shake cluster specified by the fix shake command are not all accessible to a processor. This probably means an atom has moved too far.

Shake atoms %d %d missing on proc %d at step
: The 2 atoms in a single shake cluster specified by the fix shake command are not all accessible to a processor. This probably means an atom has moved too far.

Shake cluster of more than 4 atoms
: A single cluster specified by the fix shake command can have no more than 4 atoms.

Shake clusters are connected
: A single cluster specified by the fix shake command must have a single central atom with up to 3 other atoms bonded to it.

Shake determinant = 0.0
: The determinant of the matrix being solved for a single cluster specified by the fix shake command is numerically invalid.

Shake fix must come before NPT/NPH fix
: NPT fix must be defined in input script after SHAKE fix, else the SHAKE fix contribution to the pressure virial is incorrect.

Shape command before simulation box is defined
: Self−explanatory.

Smallint setting in lmptype.h is invalid
: It has to be the size of an integer.

Smallint setting in lmptype.h is not compatible
: Smallint stored in restart file is not consistent with DSMC version you are running.

Sqrt of negative value in variable formula
: Self−explanatory.

Substitution for illegal variable
: Input script line contained a variable that could not be substituted for.

System in data file is too big
: See the setting for bigint in the src/lmptype.h file.

TAD nsteps must be multiple of t_event
: Self−explanatory.

TIP4P hydrogen has incorrect atom type
: The TIP4P pairwise computation found an H atom whose type does not agree with the specified H type.

TIP4P hydrogen is missing
: The TIP4P pairwise computation failed to find the correct H atom within a water molecule.

TMD target file did not list all group atoms
: The target file for the fix tmd command did not list all atoms in the fix group.

Tad command before simulation box is defined
: Self−explanatory.

Tagint setting in lmptype.h is invalid
: Tagint must be as large or larger than smallint.

Tagint setting in lmptype.h is not compatible
: Smallint stored in restart file is not consistent with DSMC version you are running.

Target temperature for fix nvt/npt/nph cannot be 0.0
: Self−explanatory.

Target temperature for fix rigid/nvt cannot be 0.0
: Self−explanatory.

Temper command before simulation box is defined
: The temper command cannot be used before a read_data, read_restart, or create_box command.

Temperature ID for fix bond/swap does not exist
: Self−explanatory.

Temperature ID for fix box/relax does not exist
: Self−explanatory.

Temperature ID for fix nvt/nph/npt does not exist
: Self−explanatory.

Temperature ID for fix press/berendsen does not exist
: Self−explanatory.

Temperature ID for fix temp/berendsen does not exist
: Self−explanatory.

Temperature ID for fix temp/rescale does not exist
: Self−explanatory.

Temperature control can not be used with fix nph/asphere
: Self−explanatory.

Temperature control can not be used with fix nph/sphere
: Self−explanatory.

Temperature control can not be used with fix nph
: Self−explanatory.

Temperature control must be used with fix npt/asphere
: Self−explanatory.

Temperature control must be used with fix npt/sphere
: Self−explanatory.

Temperature control must be used with fix npt
: Self−explanatory.

Temperature control must be used with fix nvt/asphere
Self−explanatory.
Temperature control must be used with fix nvt/sllod
Self−explanatory.
Temperature control must be used with fix nvt/sphere
Self−explanatory.
Temperature control must be used with fix nvt
Self−explanatory.
Temperature for fix nvt/sllod does not have a bias
The specified compute must compute temperature with a bias.
Tempering could not find thermo_pe compute
This compute is created by the thermo command. It must have been explicitly deleted by a uncompute command.
Tempering fix ID is not defined
The fix ID specified by the temper command does not exist.
Tempering temperature fix is not valid
The fix specified by the temper command is not one that controls temperature (nvt or langevin).
Thermo and fix not computed at compatible times
Fixes generate values on specific timesteps. The thermo output does not match these timesteps.
Thermo compute array is accessed out−of−range
Self−explanatory.
Thermo compute does not compute array
Self−explanatory.
Thermo compute does not compute scalar
Self−explanatory.
Thermo compute does not compute vector
Self−explanatory.
Thermo compute vector is accessed out−of−range
Self−explanatory.
Thermo custom variable cannot be indexed
Self−explanatory.
Thermo custom variable is not equal−style variable
Only equal−style variables can be output with thermodynamics, not atom−style variables.
Thermo every variable returned a bad timestep
The variable must return a timestep greater than the current timestep.
Thermo fix array is accessed out−of−range
Self−explanatory.
Thermo fix does not compute array
Self−explanatory.
Thermo fix does not compute scalar
Self−explanatory.
Thermo fix does not compute vector
Self−explanatory.
Thermo fix vector is accessed out−of−range
Self−explanatory.
Thermo keyword in variable requires lattice be defined
The xlat, ylat, zlat keywords refer to lattice properties.
Thermo keyword in variable requires thermo to use/init pe
You are using a thermo keyword in a variable that requires potential energy to be calculated, but your thermo output does not use it. Add it to your thermo output.
Thermo keyword in variable requires thermo to use/init press
You are using a thermo keyword in a variable that requires pressure to be calculated, but your thermo

output does not use it. Add it to your thermo output.

Thermo keyword in variable requires thermo to use/init temp

You are using a thermo keyword in a variable that requires temperature to be calculated, but your thermo output does not use it. Add it to your thermo output.

Thermo keyword requires lattice be defined

The xlat, ylat, zlat keywords refer to lattice properties.

Thermo style does not use press

Cannot use thermo_modify to set this parameter since the thermo_style is not computing this quantity.

Thermo style does not use temp

Cannot use thermo_modify to set this parameter since the thermo_style is not computing this quantity.

Thermo_modify int format does not contain d character

Self−explanatory.

Thermo_modify pressure ID does not compute pressure

The specified compute ID does not compute pressure.

Thermo_modify temperature ID does not compute temperature

The specified compute ID does not compute temperature.

Thermo_style command before simulation box is defined

The thermo_style command cannot be used before a read_data, read_restart, or create_box command.

This variable thermo keyword cannot be used between runs

Keywords that refer to time (such as cpu, elapsed) do not make sense in between runs.

Threshhold for an atom property that isn't allocated

A dump threshhold has been requested on a quantity that is not defined by the atom style used in this simulation.

Timestep must be >= 0

Specified timestep size is invalid.

Too big a problem to use velocity create loop all

The system size must fit in a 32−bit integer to use this option.

Too big a timestep for dump dcd

The timestep must fit in a 32−bit integer to use this dump style.

Too big a timestep for dump xtc

The timestep must fit in a 32−bit integer to use this dump style.

Too few bits for lookup table

Table size specified via pair_modify command does not work with your machine's floating point representation.

Too many atom sorting bins

This is likely due to an immense simulation box that has blown up to a large size.

Too many atoms for dump dcd

The system size must fit in a 32−bit integer to use this dump style.

Too many atoms for dump xtc

The system size must fit in a 32−bit integer to use this dump style.

Too many atoms to dump sort

Cannot sort when running with more than 2^31 atoms.

Too many exponent bits for lookup table

Table size specified via pair_modify command does not work with your machine's floating point representation.

Too many groups

The maximum number of atom groups (including the "all" group) is given by MAX_GROUP in group.cpp and is 32.

Too many iterations

You must use a number of iterations that fit in a 32−bit integer for minimization.

Too many mantissa bits for lookup table

Table size specified via pair_modify command does not work with your machine's floating point

representation.

Too many masses for fix shake

The fix shake command cannot list more masses than there are atom types.

Too many neighbor bins

This is likely due to an immense simulation box that has blown up to a large size.

Too many timesteps for NEB

You must use a number of timesteps that fit in a 32−bit integer for NEB.

Too many total atoms

See the setting for bigint in the src/lmptype.h file.

Too many total bits for bitmapped lookup table

Table size specified via pair_modify command is too large. Note that a value of N generates a 2^N size table.

Too many touching neighbors − boost MAXTOUCH

A granular simulation has too many neighbors touching one atom. The MAXTOUCH parameter in fix_shear_history.cpp must be set larger and DSMC must be re−built.

Too much per−proc info for dump

Number of local atoms times number of columns must fit in a 32−bit integer for dump.

Tree structure in joint connections

Fix poems cannot (yet) work with coupled bodies whose joints connect the bodies in a tree structure.

Triclinic box must be periodic in skewed dimensions

This is a requirement for using a non−orthogonal box. E.g. to set a non−zero xy tilt, both x and y must be periodic dimensions.

Triclinic box skew is too large

The displacement in a skewed direction must be less than half the box length in that dimension. E.g. the xy tilt must be between −half and +half of the x box length.

Tried to convert a double to int, but input_double > INT_MAX

Self−explanatory.

Two groups cannot be the same in fix spring couple

Self−explanatory.

Unbalanced quotes in input line

No matching end double quote was found following a leading double quote.

Unexpected end of data file

DSMC hit the end of the data file while attempting to read a section. Something is wrong with the format of the data file.

Units command after simulation box is defined

The units command cannot be used after a read_data, read_restart, or create_box command.

Universe/uloop variable count < # of partitions

A universe or uloop style variable must specify a number of values >= to the number of processor partitions.

Unknown command: %s

DSMCmmand is not known to DSMC. Check the input script.

Unknown identifier in data file: %s

A section of the data file cannot be read by DSMC.

Unknown table style in angle style table

Self−explanatory.

Unknown table style in bond style table

Self−explanatory.

Unknown table style in pair_style command

Style of table is invalid for use with pair_style table command.

Unrecognized lattice type in MEAM file 1

The lattice type in an entry of the MEAM library file is not valid.

Unrecognized lattice type in MEAM file 2

The lattice type in an entry of the MEAM parameter file is not valid.

Unrecognized pair style in compute pair command

Self−explanatory.

Use of compute temp/ramp with undefined lattice

Must use lattice command with compute temp/ramp command if units option is set to lattice.

Use of displace_atoms with undefined lattice

Must use lattice command with displace_atoms command if units option is set to lattice.

Use of displace_box with undefined lattice

Must use lattice command with displace_box command if units option is set to lattice.

Use of fix ave/spatial with undefined lattice

A lattice must be defined to use fix ave/spatial with units = lattice.

Use of fix deform with undefined lattice

A lattice must be defined to use fix deform with units = lattice.

Use of fix deposit with undefined lattice

Must use lattice command with compute fix deposit command if units option is set to lattice.

Use of fix dt/reset with undefined lattice

Must use lattice command with fix dt/reset command if units option is set to lattice.

Use of fix indent with undefined lattice

The lattice command must be used to define a lattice before using the fix indent command.

Use of fix move with undefined lattice

Must use lattice command with fix move command if units option is set to lattice.

Use of fix recenter with undefined lattice

Must use lattice command with fix recenter command if units option is set to lattice.

Use of fix wall with undefined lattice

Must use lattice command with fix wall command if units option is set to lattice.

Use of region with undefined lattice

If scale = lattice (the default) for the region command, then a lattice must first be defined via the lattice command.

Use of velocity with undefined lattice

If scale = lattice (the default) for the velocity set or velocity ramp command, then a lattice must first be defined via the lattice command.

Using fix nvt/sllod with inconsistent fix deform remap option

Fix nvt/sllod requires that deforming atoms have a velocity profile provided by "remap v" as a fix deform option.

Using fix nvt/sllod with no fix deform defined

Self−explanatory.

Using fix srd with inconsistent fix deform remap option

When shearing the box in an SRD simulation, the remap v option for fix deform needs to be used.

Variable evaluation before simulation box is defined

Cannot evaluate a compute or fix or atom−based value in a variable before the simulation has been setup.

Variable for compute ti is invalid style

Self−explanatory.

Variable for dump every is invalid style

Only equal−style variables can be used.

Variable for fix adapt is invalid style

Only equal−style variables can be used.

Variable for fix addforce is invalid style

Self−explanatory.

Variable for fix aveforce is invalid style

Only equal−style variables can be used.

Variable for fix efield is invalid style

Only equal−style variables can be used.

Variable for fix indent is invalid style
Only equal−style variables can be used.
Variable for fix indent is not equal style
Only equal−style variables can be used.
Variable for fix move is invalid style
Only equal−style variables can be used.
Variable for fix setforce is invalid style
Only equal−style variables can be used.
Variable for fix wall is invalid style
Only equal−style variables can be used.
Variable for fix wall/reflect is invalid style
Only equal−style variables can be used.
Variable for fix wall/srd is invalid style
Only equal−style variables can be used.
Variable for region is invalid style
Only equal−style variables can be used.
Variable for region is not equal style
Self−explanatory.
Variable for thermo every is invalid style
Only equal−style variables can be used.
Variable for velocity set is invalid style
Only atom−style variables can be used.
Variable formula compute array is accessed out−of−range
Self−explanatory.
Variable formula compute vector is accessed out−of−range
Self−explanatory.
Variable formula fix array is accessed out−of−range
Self−explanatory.
Variable formula fix vector is accessed out−of−range
Self−explanatory.
Variable name for compute atom/molecule does not exist
Self−explanatory.
Variable name for compute reduce does not exist
Self−explanatory.
Variable name for compute ti does not exist
Self−explanatory.
Variable name for dump every does not exist
Self−explanatory.
Variable name for fix adapt does not exist
Self−explanatory.
Variable name for fix addforce does not exist
Self−explanatory.
Variable name for fix ave/atom does not exist
Self−explanatory.
Variable name for fix ave/correlate does not exist
Self−explanatory.
Variable name for fix ave/histo does not exist
Self−explanatory.
Variable name for fix ave/spatial does not exist
Self−explanatory.
Variable name for fix ave/time does not exist
Self−explanatory.

Variable name for fix aveforce does not exist
Self−explanatory.
Variable name for fix efield does not exist
Self−explanatory.
Variable name for fix indent does not exist
Self−explanatory.
Variable name for fix move does not exist
Self−explanatory.
Variable name for fix setforce does not exist
Self−explanatory.
Variable name for fix store/state does not exist
Self−explanatory.
Variable name for fix wall does not exist
Self−explanatory.
Variable name for fix wall/reflect does not exist
Self−explanatory.
Variable name for fix wall/srd does not exist
Self−explanatory.
Variable name for region does not exist
Self−explanatory.
Variable name for thermo every does not exist
Self−explanatory.
Variable name for velocity set does not exist
Self−explanatory.
Variable name must be alphanumeric or underscore characters
Self−explanatory.
Velocity command before simulation box is defined
The velocity command cannot be used before a read_data, read_restart, or create_box command.
Velocity command with no atoms existing
A velocity command has been used, but no atoms yet exist.
Velocity ramp in z for a 2d problem
Self−explanatory.
Velocity temperature ID does not compute temperature
The compute ID given to the velocity command must compute temperature.
Virial was not tallied on needed timestep
You are using a thermo keyword that requires potentials to have tallied the virial, but they didn't on this timestep. See the variable doc page for ideas on how to make this work.
Wall defined twice in fix wall command
Self−explanatory.
Wall defined twice in fix wall/reflect command
Self−explanatory.
Wall defined twice in fix wall/srd command
Self−explanatory.
Weighted neighbor list values are too big
You must have less atoms per processor to use this style neighbor list.
World variable count doesn't match # of partitions
A world−style variable must specify a number of values equal to the number of processor partitions.
Write_restart command before simulation box is defined
The write_restart command cannot be used before a read_data, read_restart, or create_box command.
Zero−length lattice orient vector
Self−explanatory.

Warnings:

All element names have been set to 'C' for dump cfg
: Use the dump_modify command if you wish to override this.

Atom with molecule ID = 0 included in compute molecule group
: The group used in a compute command that operates on moleclues includes atoms with no molecule ID. This is probably not what you want.

Broken bonds will not alter angles, dihedrals, or impropers
: See the doc page for fix bond/break for more info on this restriction.

Building an occasional neighobr list when atoms may have moved too far
: This can cause DSMC to crash when the neighbor list is built. The solution is to check for building the regular neighbor lists more frequently.

Compute cna/atom cutoff may be too large to find ghost atom neighbors
: The neighbor cutoff used may not encompass enough ghost atoms to perform this operation correctly.

Computing temperature of portions of rigid bodies
: The group defined by the temperature compute does not encompass all the atoms in one or more rigid bodies, so the change in degrees−of−freedom for the atoms in those partial rigid bodies will not be accounted for.

Created bonds will not create angles, dihedrals, or impropers
: See the doc page for fix bond/create for more info on this restriction.

Dihedral problem: %d %d %d %d %d %d
: Conformation of the 4 listed dihedral atoms is extreme; you may want to check your simulation geometry.

Dump dcd/xtc timestamp may be wrong with fix dt/reset
: If the fix changes the timestep, the dump dcd file will not reflect the change.

FENE bond too long: %d %d %d %g
: A FENE bond has stretched dangerously far. It's interaction strength will be truncated to attempt to prevent the bond from blowing up.

FENE bond too long: %d %g
: A FENE bond has stretched dangerously far. It's interaction strength will be truncated to attempt to prevent the bond from blowing up.

Fix SRD walls overlap but fix srd overlap not set
: You likely want to set this in your input script.

Fix bond/swap will ignore defined angles
: See the doc page for fix bond/swap for more info on this restriction.

Fix move does not update angular momentum
: Atoms store this quantity, but fix move does not (yet) update it.

Fix move does not update quaternions
: Atoms store this quantity, but fix move does not (yet) update it.

Fix recenter should come after all other integration fixes
: Other fixes may change the position of the center−of−mass, so fix recenter should come last.

Fix srd SRD moves may trigger frequent reneighboring
: This is because the SRD particles may move long distances.

Fix srd grid size > 1/4 of big particle diameter
: This may cause accuracy problems.

Fix srd no−slip wall collisions with bin shifting
: This is an inconsistent setting in your input script.

Fix srd particle moved outside valid domain
: This may indicate a problem with your simulation parameters.

Fix srd particles may move > big particle diameter
: This may cause accuracy problems.

Fix srd viscosity < 0.0 due to low SRD density
: This may cause accuracy problems.

Fix thermal/conductivity comes before fix ave/spatial

The order of these 2 fixes in your input script is such that fix thermal/conductivity comes first. If you are using fix ave/spatial to measure the temperature profile induced by fix viscosity, then this may cause a glitch in the profile since you are averaging immediately after swaps have occurred. Flipping the order of the 2 fixes typically helps.

Fix viscosity comes before fix ave/spatial

The order of these 2 fixes in your input script is such that fix viscosity comes first. If you are using fix ave/spatial to measure the velocity profile induced by fix viscosity, then this may cause a glitch in the profile since you are averaging immediately after swaps have occurred. Flipping the order of the 2 fixes typically helps.

Group for fix_modify temp != fix group

The fix_modify command is specifying a temperature computation that computes a temperature on a different group of atoms than the fix itself operates on. This is probably not what you want to do.

Improper problem: %d %d %d %d %d %d

Conformation of the 4 listed improper atoms is extreme; you may want to check your simulation geometry.

Kspace_modify slab param < 2.0 may cause unphysical behavior

The kspace_modify slab parameter should be larger to insure periodic grids padded with empty space do not overlap.

Less insertions than requested

Less atom insertions occurred on this timestep due to the fix pour command than were scheduled. This is probably because there were too many overlaps detected.

Lost atoms: original %.15g current %.15g

A thermodynamic computation has detected lost atoms.

Mismatch between velocity and compute groups

The temperature computation used by the velocity command will not be on the same group of atoms that velocities are being set for.

More than one compute centro/atom

It is not efficient to use compute centro/atom more than once.

More than one compute cluster/atom

It is not efficient to use compute cluster/atom more than once.

More than one compute cna/atom defined

It is not efficient to use compute cna/atom more than once.

More than one compute coord/atom

It is not efficient to use compute coord/atom more than once.

More than one compute damage/atom

It is not efficient to use compute ke/atom more than once.

More than one compute ke/atom

It is not efficient to use compute ke/atom more than once.

More than one fix poems

It is not efficient to use fix poems more than once.

More than one fix rigid

It is not efficient to use fix rigid more than once.

New thermo_style command, previous thermo_modify settings will be lost

If a thermo_style command is used after a thermo_modify command, the settings changed by the thermo_modify command will be reset to their default values. This is because the thermo_modify commmand acts on the currently defined thermo style, and a thermo_style command creates a new style.

No fixes defined, atoms won't move

If you are not using a fix like nve, nvt, npt then atom velocities and coordinates will not be updated during timestepping.

No joints between rigid bodies, use fix rigid instead

The bodies defined by fix poems are not connected by joints. POEMS will integrate the body motion, but

it would be more efficient to use fix rigid.

Not using real units with pair reax

This is most likely an error, unless you have created your own ReaxFF parameter file in a different set of units.

One or more atoms are time integrated more than once

This is probably an error since you typically do not want to advance the positions or velocities of an atom more than once per timestep.

One or more compute molecules has atoms not in group

The group used in a compute command that operates on moleclues does not include all the atoms in some molecules. This is probably not what you want.

One or more respa levels compute no forces

This is computationally inefficient.

Pair COMB charge %.10f with force %.10f hit max barrier

Something is possibly wrong with your model.

Pair COMB charge %.10f with force %.10f hit min barrier

Something is possibly wrong with your model.

Pair dsmc: num_of_collisions > number_of_A

Collision model in DSMC is breaking down.

Pair dsmc: num_of_collisions > number_of_B

Collision model in DSMC is breaking down.

Particle deposition was unsuccessful

The fix deposit command was not able to insert as many atoms as needed. The requested volume fraction may be too high, or other atoms may be in the insertion region.

Reducing PPPM order b/c stencil extends beyond neighbor processor

DSMC is attempting this in order to allow the simulation to run. It should not effect the PPPM accuracy.

Replacing a fix, but new group != old group

The ID and style of a fix match for a fix you are changing with a fix command, but the new group you are specifying does not match the old group.

Replicating in a non−periodic dimension

The parameters for a replicate command will cause a non−periodic dimension to be replicated; this may cause unwanted behavior.

Resetting reneighboring criteria during PRD

A PRD simulation requires that neigh_modify settings be delay = 0, every = 1, check = yes. Since these settings were not in place, DSMC changed them and will restore them to their original values after the PRD simulation.

Resetting reneighboring criteria during TAD

A TAD simulation requires that neigh_modify settings be delay = 0, every = 1, check = yes. Since these settings were not in place, DSMC changed them and will restore them to their original values after the PRD simulation.

Resetting reneighboring criteria during minimization

Minimization requires that neigh_modify settings be delay = 0, every = 1, check = yes. Since these settings were not in place, DSMC changed them and will restore them to their original values after the minimization.

Restart file used different # of processors

The restart file was written out by a DSMC simulation running on a different number of processors. Due to round−off, the trajectories of your restarted simulation may diverge a little more quickly than if you ran on the same # of processors.

Restart file used different 3d processor grid

The restart file was written out by a DSMC simulation running on a different 3d grid of processors. Due to round−off, the trajectories of your restarted simulation may diverge a little more quickly than if you ran on the same # of processors.

Restart file used different boundary settings, using restart file values

Your input script cannot change these restart file settings.

Restart file used different newton bond setting, using restart file value

The restart file value will override the setting in the input script.

Restart file used different newton pair setting, using input script value

The input script value will override the setting in the restart file.

Restart file version does not match DSMC version

This may cause problems when reading the restart file.

Running PRD with only one replica

This is allowed, but you will get no parallel speed−up.

SRD bin shifting turned on due to small lamda

This is done to try to preserve accuracy.

SRD bin size for fix srd differs from user request

Check if the new bin size is acceptable.

SRD bins for fix srd are not cubic enough

Check if the bin shape is acceptable.

SRD particle %d started inside big particle %d on step %d bounce %d

This may not be a problem, but indicates one or more SRD particles are being left inside solute particles.

Shake determinant < 0.0

The determinant of the quadratic equation being solved for a single cluster specified by the fix shake command is numerically suspect. DSMC will set it to 0.0 and continue.

Should not allow rigid bodies to bounce off relecting walls

DSMC allows this, but their dynamics are not computed correctly.

System is not charge neutral, net charge = %g

The total charge on all atoms on the system is not 0.0, which is not valid for Ewald or PPPM.

Table inner cutoff >= outer cutoff

You specified an inner cutoff for a Coulombic table that is longer than the global cutoff. Probably not what you wanted.

Temperature for MSST is not for group all

User−assigned temperature to MSST fix does not compute temperature for all atoms. Since MSST computes a global pressure, the kinetic energy contribution from the temperature is assumed to also be for all atoms. Thus the pressure used by MSST could be inaccurate.

Temperature for NPT is not for group all

User−assigned temperature to NPT fix does not compute temperature for all atoms. Since NPT computes a global pressure, the kinetic energy contribution from the temperature is assumed to also be for all atoms. Thus the pressure used by NPT could be inaccurate.

Temperature for fix modify is not for group all

The temperature compute is being used with a pressure calculation which does operate on group all, so this may be inconsistent.

Temperature for thermo pressure is not for group all

User−assigned temperature to thermo via the thermo_modify command does not compute temperature for all atoms. Since thermo computes a global pressure, the kinetic energy contribution from the temperature is assumed to also be for all atoms. Thus the pressure printed by thermo could be inaccurate.

Too many common neighbors in CNA %d times

More than the maximum # of neighbors was found multiple times. This was unexpected.

Too many inner timesteps in fix ttm

Self−explanatory.

Too many neighbors in CNA for %d atoms

More than the maximum # of neighbors was found multiple times. This was unexpected.

Use special bonds = 0,1,1 with bond style fene/expand

Most FENE models need this setting for the special_bonds command.

Use special bonds = 0,1,1 with bond style fene

Most FENE models need this setting for the special_bonds command.

Using compute temp/deform with inconsistent fix deform remap option

Fix nvt/sllod assumes deforming atoms have a velocity profile provided by "remap v" or "remap none" as a fix deform option.

Using compute temp/deform with no fix deform defined

This is probably an error, since it makes little sense to use compute temp/deform in this case.

Using pair tail corrections with nonperiodic system

This is probably a bogus thing to do, since tail corrections are computed by integrating the density of a periodic system out to infinity.

13. Future and history

This section lists features we are planning to add to DSMC, features of previous versions of DSMC, and features of other parallel molecular dynamics codes I've distributed.

13.1 Coming attractions

The Wish list link on the DSMC WWW page gives a list of features we are hoping to add to DSMC in the future, including contact names of individuals you can email if you are interested in contributing to the developement or would be a future user of that feature.

You can also send email to the developers if you want to add your wish to the list.

13.2 Past versions

DSMC development began in the mid 1990s under a cooperative research &development agreement (CRADA) between two DOE labs (Sandia and LLNL) and 3 companies (Cray, Bristol Myers Squibb, and Dupont). The goal was to develop a large−scale parallel classical MD code; the coding effort was led by Steve Plimpton at Sandia.

After the CRADA ended, a final F77 version, DSMC 99, was released. As development of DSMC continued at Sandia, its memory management was converted to F90; a final F90 version was released as DSMC 2001.

The current DSMC is a rewrite in C++ and was first publicly released as an open source code in 2004. It includes many new features beyond those in DSMC 99 or 2001. It also includes features from older parallel MD codes written at Sandia, namely ParaDyn, Warp, and GranFlow (see below).

In late 2006 we began merging new capabilities into DSMC that were developed by Aidan Thompson at Sandia for his MD code GRASP, which has a parallel framework similar to DSMC. Most notably, these have included many−body potentials − Stillinger−Weber, Tersoff, ReaxFF − and the associated charge−equilibration routines needed for ReaxFF.

The History link on the DSMC WWW page gives a timeline of features added to the C++ open−source version of DSMC over the last several years.

These older codes are available for download from the DSMC WWW site, except for Warp &GranFlow which were primarily used internally. A brief listing of their features is given here.

DSMC 2001

- F90 + MPI
- dynamic memory
- spatial−decomposition parallelism
- NVE, NVT, NPT, NPH, rRESPA integrators
- LJ and Coulombic pairwise force fields
- all−atom, united−atom, bead−spring polymer force fields
- CHARMM−compatible force fields
- class 2 force fields

- 3d/2d Ewald &PPPM
- various force and temperature constraints
- SHAKE
- Hessian−free truncated−Newton minimizer
- user−defined diagnostics

DSMC 99

- F77 + MPI
- static memory allocation
- spatial−decomposition parallelism
- most of the DSMC 2001 features with a few exceptions
- no 2d Ewald &PPPM
- molecular force fields are missing a few CHARMM terms
- no SHAKE

Warp

- F90 + MPI
- spatial−decomposition parallelism
- embedded atom method (EAM) metal potentials + LJ
- lattice and grain−boundary atom creation
- NVE, NVT integrators
- boundary conditions for applying shear stresses
- temperature controls for actively sheared systems
- per−atom energy and centro−symmetry computation and output

ParaDyn

- F77 + MPI
- atom− and force−decomposition parallelism
- embedded atom method (EAM) metal potentials
- lattice atom creation
- NVE, NVT, NPT integrators
- all serial DYNAMO features for controls and constraints

GranFlow

- F90 + MPI
- spatial−decomposition parallelism
- frictional granular potentials
- NVE integrator
- boundary conditions for granular flow and packing and walls
- particle insertion

clear command

Syntax:

```
clear
```

Examples:

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

Description:

This command deletes all atoms, restores all settings to their default values, and frees all memory allocated by DSMC. Once a clear command has been executed, it is as if DSMC 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 (variable command).

Restrictions: none

Related commands: none

Default: none

echo command

Syntax:

```
echo style
```

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

Examples:

```
echo both
echo log
```

Description:

This command determines whether DSMC 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.

Restrictions: none

Related commands: none

Default:

```
echo log
```

if command

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 DSMC commands to execute if condition is met, each enclosed in quotes
- elif = optional word, can appear multiple times
- f1,f2,...,fN = one or more DSMC commands to execute if elif condition is met, each enclosed in quotes (optional arguments)
- else = optional argument
- e1,e2,...,eN = one or more DSMC commands to execute if no condition is met, each enclosed in quotes (optional arguments)

Examples:

```
if "${steps} > 1000" then exit
if "$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 ${eng_previous}" then "jump file
```

Description:

This command provides an in−then−else capability within an input script. A Boolean expression is evaluted 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 witht the else keyword, namely (e1, e2, ..., eN), are executed. The elif and else keywords and their associated commands are optional. If they aren't 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 DSMC 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.

IMPORTANT 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. See this section of the manual for 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 an invalid DSMC command, such as "exit" in the first example above, then executing the command will cause DSMC 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 double loop which uses the if and jump commands to break out of the inner loop when a condition is met, then continues iterating thru 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 "print 'Jumping to another script'" "jump in.script break"
  next      b
  jump      in.script loopb
label       break
variable    b delete

next        a
jump        in.script 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:

```
0.2, 100, 1.0e20, −15.4, etc
```

and Boolean operators:

```
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 a variable reference like $a or ${abc}, or another Boolean expression.

If a variable is used it must produce a number when evaluated and substituted for in the expression, else an error will be generated.

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 "||" has 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.

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. 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 NOT operator returns 1.0 if its argument is 0.0, else it returns 0.0.

The overall Boolean expression produces a TRUE result if the result is non−zero. If the result is zero, the expression result is FALSE.

Restrictions: none

Related commands:

variable, print

Default: none

include command

Syntax:

```
include file
```

- file = filename of new input script to switch to

Examples:

```
include newfile
include in.run2
```

Description:

This command opens a new input script file and begins reading DSMC 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 DSMC could run for a long time.

If the filename is a variable (see the variable command), different processor partitions can run different input scripts.

Restrictions: none

Related commands:

variable, jump

Default: none

jump command

Syntax:

```
jump file label
```

- file = filename of new input script to switch to
- label = optional label within file to jump to

Examples:

```
jump newfile
jump in.run2 runloop
jump SELF runloop
```

Description:

This command closes the current input script file, opens the file with the specified name, and begins reading DSMC 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.

IMPORTANT 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. This can be worked around by using the −in command−line argument or the −var command−line argument to pass the script name as a variable to the input script In the latter case, the "fname" variable could be used in place of SELF. E.g.

```
lmp_g++ −in in.script

lmp_g++ −var fname n.script <in.script
```

The 2nd 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, DSMC 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

variable f world script.1 script.2 script.3 script.4
jump $f
```

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 thru 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
```

Restrictions:

If you jump to a file and it does not contain the specified label, DSMC will come to the end of the file and exit.

Related commands:

variable, include, label, next

Default: none

label command

Syntax:

```
label ID
```

- ID = string used as label name

Examples:

```
label xyz
label loop
```

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.

Restrictions: none

Related commands: none

Default: none

log command

Syntax:

```
log file
```

- file = name of new logfile

Examples:

```
log log.equil
```

Description:

This command closes the current DSMC 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 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.dsmc" is the default log file for a DSMC run. The name of the initial log file can also be set by the command−line switch −log. See this section for details.

Restrictions: none

Related commands: none

Default:

The default DSMC log file is named log.dsmc

next command

Syntax:

```
next variables
```

- variables = one or more variable names

Examples:

```
next x
next a t x myTemp
```

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 DSMC 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*, *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 or values. *String*− or *atom*− or *equal*− or *world*−style variables cannot be used with the 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.

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 *universe*− or *uloop*−style variables, the next value is assigned to whichever processor partition executes the command first. All processors in the partition are assigned the same value. Running DSMC on multiple partitions of processors via the "−partition" command−line switch is described in this section of the manual. *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 such a DSMC 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 thru 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 4th 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 thru 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
```

Restrictions: none

Related commands:

jump, include, shell, variable,

Default: none

print command

Syntax:

```
print str
```

- str = text string to print, which may contain variables

Examples:

```
print "Done with equilibration"
print Vol=$v
print "The system volume is now $v"
print 'The system volume is now $v'
```

Description:

Print a text string to the screen and logfile. One line of output is generated. If the string has white space in it (spaces, tabs, etc), then you must enclose it in quotes so that it is treated as a single argument. If variables are included in the string, they will be evaluated and their current values printed.

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).

See the variable command for a description of *equal* style variables which are typically the most useful ones to use with the print command. Equal−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.

Restrictions: none

Related commands:

fix print, variable

Default: none

shell command

Syntax:

```
shell cmd args
```

- cmd = *cd* or *mkdir* or *mv* or *rm* or *rmdir* 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
  rm args = file1 file2 ...
    file1,file2 = one or more filenames to delete
  rmdir args = dir1 dir2 ...
    dir1,dir2 = one or more directories to delete
  anything else is passed as a command to the shell for direct execution
```

Examples:

```
shell cd sub1
shell cd ..
shell mkdir tmp1 tmp2 tmp3
shell rmdir tmp1
shell mv log.lammps hold/log.1
shell rm TMP/file1 TMP/file2
shell my_setup file1 10 file2
shell my_post_process 100 dump.out
```

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.

This is means to invoke other commands 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 DSMC. Or you can run a program that post−processes DSMC 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.

The *cd* cmd executes the Unix "cd" command to change the working directory. All subsequent DSMC commands that read/write files will use the new directory. All processors execute this command.

The *mkdir* cmd executes the Unix "mkdir" command to create one or more directories.

The *mv* cmd executes the Unix "mv" command to rename a file and/or move it to a new directory.

The *rm* cmd executes the Unix "rm" command to remove one or more files.

The *rmdir* cmd executes the Unix "rmdir" command to remove one or more directories. A directory must be empty to be successfully removed.

Any other cmd 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.

Restrictions:

DSMC does not detect errors or print warnings when any of these commands execute. E.g. if the specified directory does not exist, executing the *cd* command will silently do nothing.

Related commands: none

Default: none

variable command

Syntax:

```
variable name style args ...
```

- name = name of variable to define
- style = *delete* or *index* or *loop* or *world* or *universe* or *uloop* or *string* or *equal* or *atom*

```
  delete = no args
    index args = one or more strings
    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
    world args = one string for each partition of processors
    universe args = one or more strings
    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
    string arg = one string
    equal or atom args = one formula containing numbers, thermo keywords, math operations, grou
      numbers = 0.0, 100, −5.4, 2.8e−4, etc
      constants = PI
      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, xy, x>=y, xx||y, !x
      math functions = sqrt(x), exp(x), ln(x), log(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)
                       ramp(x,y), stagger(x,y), logfreq(x,y,z), vdisplace(x,y), swiggle(x,y,z),
      variable references = v_name, v_name[i]
```

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 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 x delete
```

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 be used in several ways in DSMC. A variable can be 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 *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/spatial and fix ave/atom commands).

In the discussion that follows, the "name" of the variable is the arbitrary string that is the 1st 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 1st example above, it can contain other variables as in the 2nd example, or it can be a formula as in the 3rd 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.

IMPORTANT NOTE: When the input script line that defines a variable of style *equal* or *atom* that contain a formula is encountered, the formula is NOT immediately evaluated and the result stored. See the discussion below about "Immediate Evaluation of Variables" if you want to do this.

IMPORTANT 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 2 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* and *equal* and *atom* 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.

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.

This section of the manual 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*, *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
```

For the *index* style, one or more strings are specified. Initially, the 1st 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; see this section for details.

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.

For the *world* style, one or more strings are specified. There must be one string for each processor partition or "world". See this section of the manual for information on running DSMC 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 *universe* style, one or more strings are specified. There must be at least as many strings as there are processor partitions or "worlds". See this page for information on running DSMC 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 DSMC run.

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 *equal* 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 *atom* style variables the formula computes one quantity for each atom whenever it is evaluated.

Note that *equal* 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 *atom* style variables, since there is only one string.

The formula for an *equal* 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, an formula can contain numbers, thermo keywords, math operators, math functions, group functions, region functions, atom values, atom vectors, compute references, fix references, and references to other variables.

Number	0.2, 100, 1.0e20, −15.4, etc
Constant	PI
Thermo keywords	vol, pe, ebond, etc
Math operators	(), −x, x+y, x−y, x*y, x/y, x^y, x==y, x!=y, xy, x>=y, xx||y, !x
Math functions	sqrt(x), exp(x), ln(x), log(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),
Other variables	v_name, v_name[i]

Most of the formula elements produce a scalar value. A few produce a per−atom vector of values. These are the atom vectors, compute references that represent a per−atom vector, fix references that represent a per−atom vector, and variables that are atom−style variables. Math functions that operate on scalar values produce a scalar value; math function that operate on per−atom vectors do so element−by−element and produce a per−atom vector.

A formula for equal−style variables cannot use any 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. 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 inlcuded in the formula evaluation. The variable evaluates to 0.0 for atoms not in the group.

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 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 "||" has 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.

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

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 satifsy 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 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 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.

IMPORTANT NOTE: Internally, there is just one random number generator for all equal−style variables and 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.

Variable References

Variable references access quantities calulated 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, atom−style variables generate a per−atom vector of values; all other variable styles generate a global scalar value. An equal−style variable can reference a global scalar value produced by another variable, but not a per−atom vector produced by an atom−style variable. Atom−style variables can reference either global scalar or per−atom vector values produced by kind of variable.

Examples of different kinds of variable references are as follows. There is no ambiguity as to what a reference means, since variables produce only a global scalar or a per−atom vectors, never both.

v_name	scalar, or per−atom vector
v_name[I]	atom I's value in per−atom vector

IMPORTANT NOTE: If you define variables in circular manner like this:

```
variable a equal v_b
variable b equal v_a
print $a
```

then DSMC may run for a while when the print statement is invoked!

Immediate Evaluation of Variables:

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 command, including a variable command, to force the immediate evaluation of the referenced variable and the substitution of its value into the command. The latter is a required kind of argument to some commands (e.g. the fix ave/spatial 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 preceeded by $ signs. For example,

```
variable vratio equal "${vfinal}/${v0}"
```

This is because the quotes prevent variable substitution (see this section on parsing input script commands), and thus an error will occur when the formula for "vratio" is evaluated later.

Variable Accuracy:

Obviously, DSMC attempts to evaluate variables containing formulas (*equal* 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 pressure or energy of the system, then these quantities need to be tallied during the evaluation of the interatomic potentials (pair, bond, etc) on timesteps that the variable will need the values.

DSMC keeps track of all of this during a run or energy minimization. An error will be generated if you attempt to evaluate a variable on timesteps when 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.

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. In this case, if a compute is needed to evaluate a variable (either directly or indirectly), DSMC will not invoke the compute, but it will use a value previously calculated by the compute, and can do this only if it is current. Fixes will always provide a quantity needed by a variable, but the quantity may or may not be current. This leads to one of three kinds of behavior:

(1) The variable may be evaluated accurately. If it contains references to a compute or fix, and these values were calculated on the last timestep of a preceeding run, then they will be accessed and used by the variable and the result will be accurate.

(2) DSMC may not be able to evaluate the variable and generate an error. For example, if the variable requires a quantity from a compute that is not current, DSMC will generate an error. This means, for example, that such a variable cannot be evaluated before the first run has occurred. Likewise, in between runs, such a variable cannot be accessed unless it was evaluated on the last timestep of the preceding run, e.g. by thermodynamic output.

One way to get around this problem is to perform a 0−timestep run before using the variable. For example, these commands

```
variable t equal temp
print "Initial temperature = $t"
run 1000
```

will generate an error if the run is the first run specified in the input script, because generating a value for the "t" variable requires a compute for calculating the temperature to be invoked.

However, this sequence of commands would be fine:

```
run 0
variable t equal temp
print "Initial temperature = $t"
run 1000
```

The 0−timestep run initializes and invokes various computes, including the one for temperature, so that the value it stores is current and can be accessed by the variable "t" after the run has completed. Note that a 0−timestep run does not alter the state of the system, so it does not change the input state for the 1000−timestep run that follows. Also note that the 0−timestep run must actually use and invoke the compute in question (e.g. via thermo or dump output) in order for it to enable the compute to be used in a variable after the run. Thus if you are trying to print a variable that uses a compute you have defined, you could insure it was invoked on the last timestep of the preceding run by including it in thermodynamic output.

Unlike computes, fixes will never generate an error if their values are accessed by a variable in between runs. They always return some value to the variable. However, the value may not be what you expect if the fix has not yet calculated the quantity of interest or it is not current. For example, the fix indent command stores the force on the indenter. But this is not computed until a run is performed. Thus if a variable attempts to print this value before the first run, zeroes will be output. Again, performing a 0−timestep run before printing the variable has the desired effect.

(3) The variable may be evaluated incorrectly. And DSMC may have no way to detect this has occurred. Consider the following sequence of commands:

```
pair_coeff 1 1 1.0 1.0
run 1000
pair_coeff 1 1 1.5 1.0
variable e equal pe
print "Final potential energy = $e"
```

The first run is performed using one setting for the pairwise potential defined by the pair_style and pair_coeff commands. The potential energy is evaluated on the final timestep and stored by the compute pe compute (this is done by the thermo_style command). Then a pair coefficient is changed, altering the potential energy of the system. When the potential energy is printed via the "e" variable, DSMC will use the potential energy value stored by the compute pe compute, thinking it is current. There are many other commands which could alter the state of the system between runs, causing a variable to evaluate incorrectly.

The solution to this issue is the same as for case (2) above, namely perform a 0−timestep run before the variable is evaluated to insure the system is up−to−date. For example, this sequence of commands would print a potential energy that reflected the changed pairwise coefficient:

```
pair_coeff 1 1 1.0 1.0
run 1000
pair_coeff 1 1 1.5 1.0
run 0
variable e equal pe
print "Final potential energy = $e"
```

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.

All *universe*− and *uloop*−style variables defined in an input script must have the same number of values.

Related commands:

next, jump, include, temper, fix print, print

Default: none