DSMC Users Manual

name of code

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

Table of Contents

Table of Contents

DSMC Documentation

Version info:

The DSMC "version" is the date when it was released, such as 10 Jan 2012. 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 ???.

DSMC is a Direct Simulation Montel Carlo simulator 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, and Michael Gallis_mg who can be contacted at sjplimp,magalli at sandia.gov. The DSMC WWW Site at http://www.sandia.gov/~sjplimp/dsmc.html 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.

1.1 What is DSMC

DSMC is a Direct Simulation Montel Carlo code that models rarefied gases, using a variety of collision, chemistry, boundary condition models.

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 as well as commodity 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 collision or chemistry models, boundary conditions, or diagnostics. See this section for more details.

DSMC is written in C++ which is used at a hi-level to structure the code and its options in an object-oriented fashion. The kernel computations use simple data structures and C-like code for effciency. So DSMC is really written in an object-oriented C style.

DSMC was developed with internal funding at Sandia National Laboratories, a US Department of Energy lab. 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 collision model, boundary condition, or diagnostic, 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
- 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

Collision models

(atom style command)

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://dsmc.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

- Steps to build a DSMC executable
- Common errors that can occur when making DSMC
- Additional build tips
- Building for a Mac
- Building for Windows

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:

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. Problem 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 dimension

(2) Problem definition

These items must be setup to run a DSMC calculation:

- simulation box via create_box
- grid via create_grid
- particles via create_particles

(3) Settings

Once particles and the grid are defined, a variety of settings can be specified: simulation parameters, output options, etc.

Various simulation parameters are set by these commands: timestep

Output is set by these commands:

(4) Run a simulation

A simulation is run using the run 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:

dimension

Problem definition:

create_box, create_grid, create_particles

Settings:

timestep, velocity

Actions:

run

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.

clear	create_particles	create_box	create_grid	dimension	echo
if	include	jump	label	log	next
print	run	shell	suffix	timestep	variable
velocity					

---

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.

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.

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:

10.14 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.

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&group_id=4546
```

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.

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.

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

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

create_box command

Syntax:

```
create_box xlo xhi ylo yhi zlo zhi

xlo,xhi = box bounds in the x dimension (distance units)
ylo,yhi = box bounds in the y dimension (distance units)
zlo,zhi = box bounds in the z dimension (distance units)
```

Examples:

```
create_box 0 1 0 1 0 1
create_box 0 1 0 1 -0.5 0.5
create_box 0 10.0 0 5.0 -4.0 0.0
```

Description:

Set the size of the simulation box.

For a 2d simulation, as specifed by the dimension command, *zlo* < 0.0 and *zhi* > 0.0 is required. This is so that the z dimensions straddle 0.0.

Restrictions: none

Related commands: none

Default: none

create_grid command

Syntax:

```
create_grid Nx Ny Nz keyword args ...
```

- Nx,Ny,Nz = size of grid in each dimension

- zero or more keyword/arg pairs may be appended
- keyword = *stride* or *block* or *random*

```
stride args = d1 d2 d3
    d1 = x or y or z
    d2 = x or y or z, not same as d1
    d3 = x or y or z, not same as d1 or d2
  block args = Px Py Pz
    Px,Py,Pz = # of processors in each dimension
  random args = seed
    seed = random # seed (positive integer)
```

Examples:

```
create_grid 10 10 10
create_grid 10 10 10 block 2 * *
create_grid 100 100 1 stride y x z
```

Description:

Overlay a regular *Nx* by *Ny* by *Nz* grid over the simulation domain defined by the create_box command.

For 2d simulations, *Nz* must be set to 1.

The optional *stride* and *block* and *random* keywords determine how grid cells are assigned to processors. If none are specified, then the default *block* setting is used, as listed below.

The *stride* keyword means that every Pth cell is assigned to the same processor, where P is the number of processors. E.g. if there are 100 cells and 10 processors, then the 1st processor (proc 0) will be assigned cells 1,11,21, ..., 91. The 2nd processor (proc 1) will be assigned cells 2,12,22 ..., 92. And The 10th processor (proc 9) will be assigned cells 10,20,30, ..., 100.

The arguments *d1*, *d2*, *d3* are some permutation of *x*, *y*, and *z* and determine how the N grid cells are ordered. Each of the N cells has 3 indices (I,J,K) to describe its location in the 3d grid. If *d1 d2 d3* = *x z y*, then the cells will be ordered from 1 to N with the I index varying fastest, the K index next, and the J index slowest.

The *block* keyword maps the P processors to a *Px* by *Py* by *Pz* logical grid that overlays the actual *Nx* by *Ny* by *Nz* grid. This effectively assigns a contiguous 3d sub-block of cells to each processor.

Any of the *Px*, *Py*, *Pz* parameters can be specified with an asterisk "*", in which case DSMC will choose the number of processors in that dimension. It will do this based on the size and shape of the global grid so as to minimize the surface-to-volume ratio of each processor's sub-block of cells.

The product of Px, Py, Pz must equal P, the total # of processors DSMC is running on. For a 2d simulation, Pz must equal 1. If multiple partitions are being used then P is the number of processors in this partition; see this section for an explanation of the -partition command-line switch.

Note that if you run on a large, prime number of processors P, then a grid such as 1 x P x 1 will be required, which may incur extra communication costs.

The *random* keyword means that eacn grid cell will be assigned randomly to one of the processors. This is done using a random number generator initialized with the specified *seed*. Note that in this case every processor will typically not be assigned exactly the same number of cells.

Restrictions:

This command must be used after the simulation box is defined by the create_box command.

Related commands:

create_box

Default:

The option defaults are "block * * *",

create_particles command

Syntax:

```
create_particles N seed keyword value ...
```

- N = # of particles to create
- seed = random # seed (positive integer)
- zero or more keyword/value pairs may be appended
- keyword = *loop*

  loop value = *all* or *local*

Examples:

```
create_particles 10000 187483

create_particles 10000000 49892 loop local
```

Description:

Create or add an additional *N* particles to the simulation domain.

A random number generator, initialized by the specified *seed*, is used to generate a random position for each particle within the simulation box.

The particles are initialized with a velocity = 0.0.

The *loop* keyword controls how the particles are generated in parallel.

If the setting is *all*, then every processor loops over all *N* particles. As the coordinates of each is created, each processor checks what grid cell it is in, and only stores the particle if it owns that grid cell. Thus the same set of particles are created, no matter how many processors are running the simulation

If the setting is *loop*, then each of the *P* processors generates a *N/P* itself, independent of the other processors, using its own random number generation. It only adds particles to grid cells that it owns. This is a faster way to generate a large number of particles, but means that the particles generated depend on the number of processors used (though not in a statistical sense).

Restrictions: none

Related commands: none

Default:

The option defaults are loop = all.

dimension command

Syntax:

```
dimension N
```

- N = 2 or 3

Examples:

```
dimension 2
```

Description:

Set the dimensionality of the simulation. By default DSMC runs 3d simulations, but 2d simulations can also be specified.

Restrictions:

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

Related commands: none

Default:

```
dimension 3
```

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 file1" else "jump file2"
```

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

run command

Syntax:

```
run N
```

- N = # of timesteps

Examples:

```
run 10000
run 1000000
```

Description:

Run or continue a simulation for a specified number of timesteps.

Restrictions: none

Related commands: none

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

timestep command

Syntax:

```
timestep dt
```

- dt = timestep size (time units)

Examples:

```
timestep 2.0
timestep 0.003
```

Description:

Set the timestep size for subsequent simulations.

Restrictions: none

Related commands:

run

Default:

```
timestep 1.0
```

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, group
      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, x&&y, x||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, x&&y, x||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

velocity command

Syntax:

```
velocity style args keyword value ...
```

- group-ID = ID of group of atoms whose velocity will be changed
- style = *create* or *set*

```
create args = delta seed
    delta = spread in velocity (velocity units)
    seed = random # seed (positive integer)
  set args = vx vy vz
    vx,vy,vz = velocity value (velocity units)
```

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

```
sum value = no or yes
```

Examples:

```
velocity create 0.3 598093
velocity set 2.0 0.0 0.0 sum yes
```

Description:

Set or change the velocities of all the current particles in one of several styles. For each style, there are required arguments and optional keyword/value parameters. Not all options are used by each style. Each option has a default as listed below.

The *create* style generates a spread of velocities using a random number generator intialized with the specified *seed*. The *delta* value means that each velocity component from -delta to delta will sampled using a uniform random number.

The *set* style sets the velocities of all particles to the specified values.

The keyword/value option pairs are used in the following ways by the various styles.

The *sum* option is used by all styles. The new velocities will be added to the existing ones if sum = yes, or will replace them if sum = no.

Restrictions:

When using the *set* style for 2d simulations, *vz* must be specified as 0.0.

Related commands: none

Default:

The option defaults are sum = no.