SPPARKS Users Manual

Stochastic Parallel PARticle Kinetic Simulator

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

Table of Contents

Table of Contents

Table of Contents

SPPARKS Documentation

(11 Nov 2009 version of SPPARKS)

SPPARKS stands for Stochastic Parallel PARticle Kinetic Simulator.

SPPARKS is a kinetic Monte Carlo (KMC) code designed to run efficiently on parallel computers using both KMC and Metropolis Monte Carlo algorithms. 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 developers of SPPARKS are Steve Plimpton, Aidan Thompson, and Alex Slepoy. They can be contacted at sjplimp@sandia.gov, athomps@sandia.gov, and alexander.slepoy@nnsa.doe.gov. The SPPARKS WWW Site at http://www.cs.sandia.gov/~sjplimp/spparks.html has more information about the code and its uses.

The SPPARKS 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 SPPARKS documentation.

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

PDF file of the entire manual, generated by htmldoc

1. Introduction

These sections provide an overview of what SPPARKS can do, describe what it means for SPPARKS to be an open−source code, and acknowledge the funding and people who have contributed to SPPARKS.

1.1 What is SPPARKS

SPPARKS is a Monte Carlo code that has algorithms for kinetic Monte Carlo (KMC), rejection KMC (rKMC), and Metropolis Monte Carlo (MMC). On−lattice and off−lattice applications with spatial sites on which "events" occur can be simulated in parallel.

KMC is also called true KMC or rejection−free KMC. rKMC is also called null−event MC. In a generic sense the code's KMC and rKMC solvers catalog a list of events, each with an associated probability, choose a single event to perform, and advance time by the correct amount. Events may be chosen individually at random, or a sweep of enumarated sites can be performed to select possible events in a more ordered fashion.

Note that rKMC is different from Metropolis MC, which is sometimes called thermodynamic−equilibrium MC or barrier−free MC, in that rKMC still uses rates to define events, often associated with the rate for the system to cross some energy barrier. Thus both KMC and rKMC track the dynamic evolution of a system in a time−accurate manner as events are performed. Metropolis MC is typically used to sample states from a system in equilibrium or to drive a system to equilibrium (energy minimization). It does this be performing (possibly) non−physical events. As such it has no requirement to sample events with the correct relative probabilities or to limit itself to physical events (e.g. it can change an atom to a new species). Because of this it also does not evolve the system in a time−accurate manner; in general there is no "time" associated with Metropolis MC events.

Applications are implemented in SPPARKS which define events and their probabilities and acceptance/rejection criteria. They are coupled to solvers or sweepers to perform KMC or rKMC simulations. The KMC or rKMC options for an application in SPPARKS can be written to define rates based on energy differences between the initial and final state of an event and a Metropolis−style accept/reject criterion based on the Boltzmann factor SPPARKS will then perform a Metropolis−style Monte Carlo simulation.

In parallel, a geometric partitioning of the simulation domain is performed. Sub−partitioning of processor domains into colors or quadrants (2d) and octants (3d) is done to enable multiple events to be performed on multiple processors simultaneously. Communication of boundary information is performed as needed.

Parallelism can also be invoked to perform multiple runs on a collection of processors, for statistical puposes.

SPPARKS is designed to be easy to modify and extend. For example, new solvers and sweeping rules can be added, as can new applications. Applications can define new commands which are read from the input script.

SPPARKS is written in C++. It runs on single−processor desktop or laptop machines, but for some applications, can also run on parallel computers. SPPARKS will run on any parallel machine that compiles C++ and supports the MPI message−passing library. This includes distributed− or shared−memory machines.

SPPARKS is a freely−available open−source code. See the SPPARKS WWW Site for download information. It is distributed under the terms of the GNU Public License, which means you can use or modify the code however you wish. The only restrictions imposed by the GPL are on how you distribute the code further. See this section for a brief discussion of the open−source philosophy.

1.2 SPPARKS features

These are three kinds of applications in SPPARKS:

- on−lattice
- off−lattice
- general

On−lattice applications define static event sites with a fixed neighbor connectivity. Off−lattice applications define mobile event sites such as particles. A particle's neighbors are typically specified by a cutoff distance. General applications have no spatial component.

The set of on−lattice applications currently in SPPARKS are:

- diffusion model
- Ising model
- Potts model
- membrane model

The set of off−lattice applications currently in SPPARKS are:

- Metropolis atomic relaxation model

The set of general applications currently in SPPARKS are:

- biochemcial reaction network model
- test driver for solvers using a synthetic biochemical network

These are the KMC solvers currently available in SPPARKS and their scaling properties:

- linear search, O(N)
- tree search, O(logN)
- composition−rejection search, O(1)

Pre− and post−processing:

Our group has written and released a separate toolkit called Pizza.py which provides tools which can be used to setup, analyze, plot, and visualize data for SPPARKS simulations. Pizza.py is written in Python and is available for download from the Pizza.py WWW site.

1.4 Open source distribution

SPPARKS 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 SPPARKS distribution.

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

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

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

(3) If you distribute any code that used SPPARKS source code, including calling it as a library, then that must also be open−source, meaning you distribute its source code under the terms of the GPL.

(4) If you give SPPARKS 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, if you use SPPARKS for something useful or if you fix a bug or add a new feature or applicaton to the code, let us know. We would like to include your contribution in the released version of the code and/or advertise your success on our WWW page.

1.4 Acknowledgments and citations

SPPARKS is distributed by Sandia National Laboratories. SPPARKS development has been funded by the US Department of Energy (DOE), through its LDRD and ASC programs.

The primary authors of SPPARKS are Steve Plimpton, Aidan Thompson, and Alex Slepoy. They can be contacted via email: sjplimp@sandia.gov, athomps@sandia.gov, alexander.slepoy@nnsa.doe.gov.

The following Sandians have also contributed to the design and ideas in SPPARKS:

- Corbett Battaile
- Liz Holm
- Ed Webb

2. Getting Started

This section describes how to unpack, make, and run SPPARKS.

2.1 What's in the SPPARKS distribution

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

```
gunzip spparks*.tar.gz
tar xvf spparks*.tar
```

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

README	text file
LICENSE	the GNU General Public License (GPL)
doc	documentation
examples	test problems
src	source files

2.2 Making SPPARKS

Read this first:

Building SPPARKS can be non−trivial. You will likely need to edit a makefile, there are compiler options, an MPI library can be used, 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.

Building a SPPARKS executable:

The src directory contains the C++ source and header files for SPPARKS. It also contains a top−level Makefile and a MAKE sub−directory with low−level Makefile.* files for several 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
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 typically build SPPARKS more quickly.

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

Errors that can occur when making SPPARKS:

(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 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 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 SPPARKS.

(2) Other errors typically occur because the low−level Makefile isn't setup correctly for your machine. If your platform is named "foo", you need to create a Makefile.foo in the MAKE sub−directory. Use whatever existing file is closest to your platform as a starting point. See the next section for more instructions.

Editing a new low−level Makefile.foo:

These are the issues you need to address when editing a low−level Makefile for your machine. With a couple exceptions, the only portion of the file you should need to edit is the "System−specific Settings" section.

(1) Change the first line of Makefile.foo to include the word "foo" and whatever other options you set. This is the line you will see if you just type "make".

(2) Set the paths and flags for your C++ compiler, including optimization flags. You can use g++, the open−source GNU compiler, which is available on all Unix systems. Vendor compilers often produce faster code. On boxes with Intel CPUs, we suggest using the free Intel icc compiler, which you can download from Intel's compiler site.

(3) If you want SPPARKS to run in parallel, you must have an MPI library installed on your platform. If you do not use "mpicc" as your compiler/linker, then Makefile.foo needs to specify where the mpi.h file (−I switch) and the libmpi.a library (−L switch) is found. If you are installing MPI yourself, we recommend Argonne's MPICH 1.2 or 2.0 which can be downloaded from the Argonne MPI site. OpenMPI 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, so find out how to build and link with it. If you use MPICH or OpenMPI, 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 SPPARKS build, which can avoid problems that may arise when linking SPPARKS to the MPI library.

(4) If you just want SPPARKS to run on a single processor, you can use the STUBS library in place of MPI, since you don't need an MPI library installed on your system. See the Makefile.serial file for how to specify the −I and −L switches. You will also need to build the STUBS library for your platform before making SPPARKS itself. From the STUBS dir, type "make" and it will hopefully create a libmpi.a suitable for linking to SPPARKS. If the build fails, you will need to edit the STUBS/Makefile for your platform.

The file STUBS/mpi.cpp has a CPU timer function 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 SPPARKS simulations.

(5) The DEPFLAGS setting is how the C++ compiler creates a dependency file for each 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 (a long list of errors involving *.d files), then you'll need to create a Makefile.foo patterned after Makefile.tflop, which uses different rules that do not involve dependency files.

(6) There is a −D compiler switches you can set as part of CCFLAGS. The dump command will read/write gzipped files if you compile with −DSPPARKS_GZIP. It requires that your Unix support the "popen" command.

That's it. Once you have a correct Makefile.foo and you have pre−built the MPI library it uses, all you need to do from the src directory is type one of these 2 commands:

```
make foo
gmake foo
```

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

Additional build tips:

(1) Building SPPARKS for multiple platforms.

You can make SPPARKS 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" will delete all *.o object files created when SPPARKS is built.

(3) Building for a Macintosh.

OS X is BSD Unix, so it already works. See the Makefile.mac file.

2.3 Making SPPARKS with optional packages

NOTE: this sub−section is currently a placeholder. There are no packages distributed with the current version of SPPARKS.

The source code for SPPARKS is structured as a large set of core files which are always used, plus optional packages, which are groups of files that enable a specific set of features. You can see the list of both standard and user−contributed packages by typing "make package".

Any or all packages can be included or excluded when SPPARKS is built. You may wish to exclude certain packages if you will never run certain kinds of simulations.

By default, SPPARKS includes no packages.

Packages are included or excluded by typing "make yes−name" or "make no−name", where "name" is the name of the package. You can also type "make yes−all" or "make no−all" to include/exclude all packages. These commands work by simply moving files back and forth between the main src directory and sub−directories with the package name, so that the files are seen or not seen when SPPARKS is built. After you have included or excluded a package, you must re−build SPPARKS.

Additional make options exist to help manage SPPARKS 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 SPPARKS files or have downloaded a patch from the SPPARKS WWW site. Typing "make package−update" will overwrite src files with

files from the package directories if the package has been included. It should be used after a patch is installed, since patches only update the master package version of a file. Typing "make package−overwrite" will overwrite files in the package directories with src files. Typing "make package−check" will list differences between src and package versions of the same files.

2.4 Building SPPARKS as a library

SPPARKS can be built as a library, which can then be called from another application or a scripting language. Building SPPARKS as a library is done by typing

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

where foo is the machine name. 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 SPPARKS 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 libspk_foo.a which another application can link to.

When used from a C++ program, the library allows one or more SPPARKS objects to be instantiated. All of SPPARKS is wrapped in a SPPARKS_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 library.cpp and library.h.

You can add as many functions as you wish to library.cpp and library.h. In a general sense, those functions can access SPPARKS data and return it to the caller or set SPPARKS data values as specified by the caller. These 4 functions are currently included in library.cpp:

```
void spparks_open(int, char **, MPI_Comm, void **ptr);
void spparks_close(void *ptr);
int spparks_file(void *ptr, char *);
int spparks_command(void *ptr, char *);
```

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

The SPPARKS_close() function is used to shut down SPPARKS and free all its memory. The SPPARKS_file() and SPPARKS_command() functions are used to pass a file or string to SPPARKS as if it were an input file or single command read from an input script.

2.5 Running SPPARKS

By default, SPPARKS runs by reading commands from stdin; e.g. spk_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 SPPARKS on any of the sample inputs provided in the examples 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 the Potts model tests on a Linux box, using mpirun to launch a parallel job:

```
cd src
make linux
cp spk_linux ../examples/lj
cd ../examples/potts
mpirun −np 4 spk_linux <in.potts
```

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

Note that this sequence of commands copies the SPPARKS executable (spk_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 spk_linux on its own and not under mpirun). If that happens, SPPARKS 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 SPPARKS 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 SPPARKS can or can't detect, a list of all ERROR and WARNING messages, and what to do about them.

SPPARKS can run a problem on any number of processors, including a single processor. SPPARKS 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, SPPARKS recognizes several optional command−line switches which may be used in any order. For example, spk_ibm might be launched as follows:

```
mpirun −np 16 spk_ibm −var f tmp.out −log my.log −screen none <in.alloy
```

These are the command−line options:

```
−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.

```
−partition 8x2 4 5 ...
```

Invoke SPPARKS in multi−partition mode. When SPPARKS is run on P processors and this switch is not used, SPPARKS 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.

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.

`-in file`

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

`-log file`

Specify a log file for SPPARKS to write status information to. In one-partition mode, if the switch is not used, SPPARKS writes to the file log.spparks. If this switch is used, SPPARKS writes to the specified file. In multi-partition mode, if the switch is not used, a log.SPPARKS file is created with hi-level status information. Each partition also writes to a log.SPPARKS.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.

`-screen file`

Specify a file for SPPARKS to write its screen information to. In one-partition mode, if the switch is not used, SPPARKS writes to the screen. If this switch is used, SPPARKS 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.

`-var name value`

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}). The value can be any string. Using this command-line option is equivalent to putting the line "variable name index value" at the beginning of the input script. Defining a variable as a command-line argument overrides any setting for the same variable in the input script, since variables cannot be re-defined. See the variable command for more info on defining variables and this section for more info on using variables in input scripts.

3. Commands

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

3.1 SPPARKS input script

SPPARKS executes by reading commands from a input script (text file), one line at a time. When the input script ends, SPPARKS exits. Each command causes SPPARKS 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) SPPARKS 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:

```
count    ligand 10000
run      100
run      100
```

does something different than this sequence:

```
run      100
count    ligand 10000
run      100
```

In the first case, the count of ligand molecules is set to 10000 before the first simulation and whatever the count becomes will be used as input for the second simulation. In the 2nd case, the default count of 0 is used for the 1st simulation and then the count is set to 10000 molecules before the second simulation.

(2) Some commands are only valid when they follow other commands. For example you cannot set the count of a molecular species until the add_species command has been used to define that species.

(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.

(4) Some commands are only used by a specific application(s).

Many input script errors are detected by SPPARKS 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. SPPARKS 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 SPPARKS:

(1) If the line ends with a ""character (with no trailing whitespace), 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.

(3) The line is searched repeatedly for $ characters which indicate variables that are replaced with a text string. 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 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 scripts.

(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) Text with spaces can be enclosed in double quotes so it will be treated as a single argument. See the dump modify or fix print commands for examples. A '#' or '$' character that in text between double quotes will not be treated as a comment or substituted for as a variable.

3.3 Input script structure

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

A SPPARKS input script typically has 3 parts:

- choice of application, solver, sweeper
- settings
- 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 3 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) Choice of application, solver, sweep method

Use the app_style, solve_style, and sweep commands to setup the kind of simulation you wish to run. Note that sweeping is only relevant to applications that define a geometric lattice of event sites and only if you wish to perform rejection kinetic Monte Carlo updates.

(2) Settings

Parameters for a simulation can be defined by application−specific commands or by generic commands that are common to many kinds of applications. See the doc pages for individual applications for information on the former. Examples of the latter are the stats and temperature commands.

The diag_style command can also be used to setup various diagnostic computations to perform during a simulation.

(3) Run a simulation

A kinetic or Metropolis Monte Carlo simulation is performed using the run command.

3.4 Commands listed by category

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

Initialization commands:

app_style, create_box, create_sites, processors, read_sites, region, solve_style

Setting commands:

dimension, lattice, pair_coeff, pair_style, reset_time, sector, seed, sweep, set

Application−specific commands:

add_reaction, add_species, barrier, count, deposition, ecoord, inclusion, pin, temperature, volume

Output commands:

diag_style, dump, dump_modify, dump_one, stats, undump

Actions:

run,

Miscellaneous:

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

---

3.5 Individual commands

This section lists all SPPARKS 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 commands are only usable with certain applications. Also, some style options for some commands are part of specific SPPARKS packages, which means they cannot be used unless the package was included when SPPARKS was built. Not all packages are included in a default SPPARKS build. These dependencies are listed as Restrictions in the command's documentation.

add_reaction	add_species	app_style	barrier	clear	count
create_box	create_sites	deposition	diag_style	dimension	dump
dump_modify	dump_one	echo	ecoord	if	include
inclusion	jump	label	lattice	log	next
pair_coeff	pair_style	pin	print	processors	read_sites
region	reset_time	run	sector	seed	set
shell	solve_style	stats	sweep	temperature	undump
variable	volume				

Application styles. See the app_style command for one−line descriptions of each style or click on the style itself for a full description:

chemistry	diffusion	ising	ising/single	membrane	potts	potts/neigh	potts/neighonly
potts/pin	relax	test/group					

Solve styles. See the solve_style command for one−line descriptions of each style or click on the style itself for a full description:

group	linear	tree

Pair styles. See the pair_style command for one−line descriptions of each style or click on the style itself for a full description:

lj/cut

Diagnostic styles. See the diag_style command for one−line descriptions of each style or click on the style itself for a full description:

cluster	diffusion	energy	eprof	propensity

4. How−to discussions

The following sections describe how to perform various operations in SPPARKS.

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

4.1 Running multiple simulations from one input script

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

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

```
app_style ising/2d/4n 100 100 12345
...
run 1.0
run 1.0
run 1.0
run 1.0
run 1.0
```

would run 5 successive simulations of the same system for a total of 5.0 seconds of elapsed time.

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

```
app_style ising/2d/4n 100 100 12345
...
run 1.0
clear
app_style ising/2d/4n 200 200 12345
...
run 1.0
```

would run 2 independent simulations, one after the other.

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

```
variable d index run1 run2 run3 run4 run5 run6 run7 run8
shell cd $d
app_style ising/2d/4n 100 100 12345
include temperature.txt
run 1.0
shell cd ..
clear
next d
jump in.runs
```

would run 8 simulations in different directories, using a temperature.txt file in each directory with an input command to set the temperature. The same concept could be used to run the same system at 8 different sizes, using a size variable and storing the output in different log files, for example

```
variable a loop 8
variable size index 100 200 400 800 1600 3200 6400 10000
log log.${size}
app_style ising/2d/4n ${size} ${size} 12345
run 1.0
next size
next a
jump in.runs
```

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

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

4.2 Coupling SPPARKS to other codes

SPPARKS is designed to allow it to be coupled to other codes. For example, an atomistic code might relax atom positions and pass those positions to SPPARKS. Or a continuum finite element (FE) simulation might use a Monte Carlo relaxation to formulate a boundary condition on FE nodal points, compute a FE solution, and return the results to the MC calculation.

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

(1) Define a new diag_style command that calls the other code. In this scenario, SPPARKS is the driver code. During its timestepping, the diagnostic is invoked, and can make library calls to the other code, which has been linked to SPPARKS as a library. See this section of the documentation for info on how to add a new diagnostic to SPPARKS.

(2) Define a new SPPARKS command that calls the other code. This is conceptually similar to method (1), but in this case SPPARKS and the other code are on a more equal footing. Note that now the other code is not called during the even loop of a SPPARKS run, but between runs. The SPPARKS input script can be used to alternate SPPARKS runs with calls to the other code, invoked via the new command.

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

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

(3) Use SPPARKS as a library called by another code. In this case the other code is the driver and calls SPPARKS as needed. Or a wrapper code could link and call both SPPARKS and another code as libraries.

This section of the documentation describes how to build SPPARKS as a library. Once this is done, you can interface with SPPARKS either via C++, C, or Fortran (or any other language that supports a vanilla C−like interface, e.g. a scripting language). For example, from C++ you could create one (or more) "instances" of SPPARKS, pass it an input script to process, or execute individual commands, all by invoking the correct class methods in SPPARKS. From C or Fortran you can make function calls to do the same things. Library.cpp and library.h contain such a C interface with the functions:

```
void spparks_open(int, char **, MPI_Comm, void **);
void spparks_close(void *);
void spparks_file(void *, char *);
char *spparks_command(void *, char *);
```

The functions contain C++ code you could write in a C++ application that was invoking SPPARKS directly. Note that SPPARKS classes are defined within a SPPARKS namespace (SPPARKS_NS) if you use them from another C++ application.

Two of the routines in library.cpp are of particular note. The SPPARKS_open() function initiates SPPARKS and takes an MPI communicator as an argument. It returns a pointer to a SPPARKS "object". As with C++, the SPPARKS_open() function can be called multiple times, to create multiple instances of SPPARKS.

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

Library.cpp contains a SPPARKS_command() function to which the caller passes a single SPPARKS command (a string). Thus the calling code can read or generate a series of SPPARKS commands (e.g. an input script) one line at a time and pass it thru the library interface to setup a problem and then run it.

A few other sample functions are included in library.cpp, but the key idea is that you can write any functions you wish to define an interface for how your code talks to SPPARKS and add them to library.cpp and library.h. The routines you add can access any SPPARKS data. The examples/couple directory has example C++ and C codes which show how a stand−alone code can link SPPARKS as a library, run SPPARKS on a subset of processors, grab data from SPPARKS, change it, and put it back into SPPARKS.

5. Example problems

The SPPARKS distribution includes an examples sub−directory with several sample problems. Each problem is in a sub−directory of its own. Most are small models that can be 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. 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.potts.foo.P means it ran on P processors of machine "foo".

In some cases, the dump files produced by the example runs can be animated using the various visuzlization tools, such as the Pizza.py toolkit referenced in the Additional Tools section of the SPPARKS documentation. Animations of some of these examples can be viewed on the Movies section of the SPPARKS WWW Site.

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

groups	test of group−based KMC solver
ising	standard Ising model
membrane	membrane model of pore formation around protein inclusions
potts	multi−state Potts model for grain growth

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

```
cd examples/potts
cp ../../src/spk_linux .           # copy SPPARKS executable to this dir
spk_linux <in.potts               # run the problem
```

Running the simulation produces the files *dump.potts* and *log.spparks*.

6. Performance &scalability

Eventually this section will highlight SPPARKS performance in serial and parallel on interesting Monte Carlo benchmarks.

7. Additional tools

SPPARKS is designed to be a Monte Carlo (MC) kernel for performing kinetic MC or Metropolis MC computations. Additional pre− and post−processing steps are often necessary to setup and analyze a simulation. This section describes additional tools that may be useful.

Users can extend SPPARKS by writing diagnostic classes that perform desired analysis or computations. See this section for more info.

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

8. Modifying &extending SPPARKS

SPPARKS is designed in a modular fashion so as to be easy to modify and extend with new functionality.

In this section, changes and additions users can make are listed along with minimal instructions. If you add a new feature to SPPARKS 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 SPPARKS.

The best way to add a new feature is to find a similar feature in SPPARKS 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 SPPARKS 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. 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 SPPARKS to invoke the new class is as simple as adding two lines to the style_user.h file, in the same syntax as other SPPARKS classes are specified in the style.h file.

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 SPPARKS more complex or cause side−effect bugs.

Here is a concrete example. Suppose you write 2 files app_foo.cpp and app_foo.h that define a new class AppFoo that implements a Monte Carlo model described in the classic 1997 paper by Foo, et al. If you wish to invoke that application in a SPPARKS input script with a command like

```
app_style foo 0.1 3.5
```

you put your 2 files in the SPPARKS src directory and re−make the code. The app_foo.h file should have these lines at the top

```
#ifdef APP_CLASS
AppStyle(foo,AppFoo)
#else
```

where "foo" is the style keyword to be used in the app_style command, and AppFoo is the class name in your C++ files.

When you re−make SPPARKS, your new application becomes part of the executable and can be invoked with a app_style command like the example above. Arguments like 0.1 and 3.5 can be defined and processed by your new class.

Here is a list of the new features that can be added in this way.

- Application styles
- Diagnostic styles
- Input script commands
- Solve styles

As illustrated by the application example, these options are referred to in the SPPARKS 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 SPPARKS. 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 SPPARKS expects. Virtual functions that are not set to 0 are functions you can optionally define.

Application styles

In SPPARKS, applications are what define the simulation model that is evolved via Monte Carlo algorithms. A new model typically requires adding a new application to the code. Read the doc page for the app_style command to understand the distinction between on−lattice and off−lattice applications. A new off−lattice application can be anything you wish. On−lattice applications are derive from the AppLattice class.

For on−lattice and off−lattice applications, here is a brief description of methods you define in your new derived class. Some of them are required; some are optional. See app.h for details.

input_app	additional commands the application defines
grow_app	set pointers to per−site arrays used by the application
init_app	initialize the application before a run
site_energy	compute energy of a site
site_event_rejection	peform an event with null−bin rejection (for rKMC)
site_propensity	compute propensity of all events on a site (for KMC)
site_event	perform an event (for KMC)

Note that two of the methods are required if you want your application to perform kinetic Monte Carlo (KMC) with a solver. One of the methods is required if you want your application to perform rejection KMC (rKMC) with a sweep method.

The constructor for your application class also needs to define, to insure proper operation with the "KMC solvers'_solve.html and rejection KMC sweep methods. These are the flags, all of which have default values set in app_lattice.cpp:

ninteger	how many integer values are defined per site
ndouble	how many floating point values are defined per site
delpropensity	how many neighbors away values are needed to compute propensity
delevent	how many neighbors away may the value can be changed by an event
allow_kmc	1 if methods are provided for KMC
allow_rejection	1 if methods are provided for rejection KMC
allow_masking	1 if rKMC method supports masking
numrandom	# of random numbers used by the site_event_rejection method

Diagnostic styles

Diagnostic classes compute some form of analysis periodically during a simulation. See the diag_style command for details.

To add a new diagnostic, here is a brief description of methods you define in your new derived class. Some of them are required; some are optional. See diag.h for details.

init	setup the computation
compute	perform the analysis computation
stats_header	what to add to statistics header for this diagnostic
stats	fields added to statistics by this diagnostic

Input script commands

New commands can be added to SPPARKS input scripts by adding new classes that have a "command" method and are listed in the Command sections of style_user.h (or style.h). For example, the shell commands (cd, mkdir, rm, etc) are implemented in this fashion. When such a command is encountered in the SPPARKS input script, SPPARKS simply creates a class with the corresponding name, invokes the "command" method of the class, and passes it the arguments from the input script. The command method can perform whatever operations it wishes on SPPARKS data structures.

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

command	operations performed by the new command

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

Solve styles

In SPPARKS, a solver performs the kinetic Monte Carlo (KMC) operation of selecting an event from a list of events and associated probabilities. See the solve_style command for details.

To add a new KMC solver, here is a brief description of methods you define in your new derived class. Some of them are required; some are optional. See diag.h for details.

Here is a brief description of methods you define in your new derived class. All of them are required. See solve.h for details.

clone	make a copy of the solver for use within a sector of the domain
init	initialize the solver
update	update one or more event probabilities
resize	change the number of events in the list
event	select an event and associated timestep

(Foo) Foo, Morefoo, and Maxfoo, J of Classic Monte Carlo Applications, 75, 345 (1997).

9. Errors

This section describes the various kinds of errors you can encounter when using SPPARKS.

9.1 Common problems

A SPPARKS simulation typically has two stages, setup and run. Many SPPARKS errors are detected at setup time; others may not occur until the middle of a run.

SPPARKS tries to flag errors and print informative error messages so you can fix the problem. Of course SPPARKS cannot figure out your physics mistakes, like choosing too big a timestep or setting up an invalid lattice. If you find errors that SPPARKS 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.spparks file or using the echo command to see it on the screen. For example you can run your script as

```
spk_linux −echo screen <in.script
```

For a given command, SPPARKS expects certain arguments in a specified order. If you mess this up, SPPARKS will often flag the error, but it may read a bogus argument and assign a value that is not what you wanted. E.g. if the input parser reads the string "abc" when expecting an integer value, it will assign the value of 0 to a variable.

Generally, SPPARKS will print a message to the screen and exit gracefully when it encounters a fatal error. Sometimes it will print a WARNING and continue on; you can decide if the WARNING is important or not. If SPPARKS 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:

SPPARKS runs in the available memory each processor can allocate. 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 when one of these small requests is made, in which case the code will crash, since SPPARKS doesn't trap on those errors.

Illegal arithmetic can cause SPPARKS to run slow or crash. This is typically due to invalid physics and numerics that your simulation is computing. If you see wild energy values or NaN values in your SPPARKS output, something is wrong with your simulation.

In parallel, one way SPPARKS 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.

9.2 Reporting bugs

If you are confident that you have found a bug in SPPARKS, please send an email to the developers.

First, check the "New features and bug fixes" section of the SPPARKS WWW site to see if the bug has already been reported or fixed.

If not, the most useful thing you can do for us is to isolate the problem. Run it on the smallest problem and fewest number of processors and with the simplest input script that reproduces the bug.

In your email, describe the problem and any ideas you have as to what is causing it or where in the code the problem might be. We'll request your input script and data files if necessary.

9.3 Error &warning messages

These are two alphabetic lists of the ERROR and WARNING messages SPPARKS prints out and the reason why. If the explanation here is not sufficient, the documentation for the offending command may help. Grepping the source files for the text of the error message and staring at the source code and comments is also not a bad idea! Note that sometimes the same message can be printed from multiple places in the code.

Errors:

Adding site to bin it is not in
: Internal SPPARKS error.

Adding site to illegal bin
: Internal SPPARKS error.

All pair coeffs are not set
: Self−explanatory.

All universe/uloop variables must have same # of values
: Self−explanatory.

All variables in next command must be same style
: Self−explanatory.

Another input script is already being processed
: Cannot attempt to open a 2nd input script, when the original file is still being processed.

App cannot use both a KMC and rejection KMC solver
: You cannot define both a solver and sweep option.

App did not set dt_sweep
: Internal SPPARKS error.

App needs a KMC or rejection KMC solver
: You must define either a solver or sweep option.

App relax requires a pair potential
: Self−explanatory.

App style proc count is not valid for 1d simulation
: There can only be 1 proc in y and z dimensions for 1d models.

App style proc count is not valid for 2d simulation
: There can only be 1 proc in the z dimension for 2d models.

App_style command after simulation box is defined
: Self−explanatory.

App_style specific command before app_style set
: Self−explanatory.

Application cutoff is too big for processor sub−domain
: There must be at least 2 bins per processor in each dimension where sectoring occurs.

Arccos of invalid value in variable formula
: Argument of arccos() must be between −1 and 1.

Arcsin of invalid value in variable formula
: Argument of arcsin() must be between −1 and 1.

Bad connectivity result
: Internal SPPARKS error. Should not occur.

Box bounds are invalid
: Lo bound >= hi bound.

Can only read neighbors for on−lattice applications
: Self−explanatory.

Can only use ecoord command with app_style diffusion nonlinear
: Self−explanatory.

Cannot color this combination of lattice and app
: Coloring is not supported on this lattice for the neighbor dependencies of this application.

Cannot color wihtout a lattice definition of sites
: Self−explanatory.

Cannot create box after simulation box is defined
: Self−explanatory.

Cannot create box with this application style
: This application does not support spatial domains.

Cannot create sites after sites already exist
: Self−explanatory.

Cannot create sites with undefined lattice
: Must use lattice commands first to define a lattice.

Cannot define Schwoebel barrier without Schwoebel model
: Self−explanatory.

Cannot open diag style cluster dump file
: Self−explanatory.

Cannot open diag_style cluster dump file
: Self−explanatory.

Cannot open diag_style cluster output file
: Self−explanatory.

Cannot open dump file
: Self−explanatory.

Cannot open file %s
: Self−explanatory.

Cannot open gzipped file
: Self−explantory.

Cannot open input script %s
: Self−explanatory.

Cannot open log.spparks
: Self−explanatory.

Cannot open logfile %s
: Self−explanatory.

Cannot open logfile
: Self−explanatory.

Cannot open screen file
: The screen file specified as a command−line argument cannot be opened. Check that the directory you are running in allows for files to be created.

Cannot open universe log file
: For a multi−partition run, the master log file cannot be opened. Check that the directory you are running in allows for files to be created.

Cannot open universe screen file
: For a multi−partition run, the master screen file cannot be opened. Check that the directory you are running in allows for files to be created.

Cannot read neighbors unless max neighbors is set
: This is a setting in the header of the sites file.

Cannot read sites after sites already exist
: Self−explanatory.

Cannot redefine variable as a different style
: An equal−style variable can be re−defined but only if it was originally an equal−style variable.

Cannot run application until simulation box is defined
: Self−explanatory.

Cannot use %s command until sites exist
: This command requires sites exist before using it in an input script.

Cannot use KMC solver in parallel with no sectors
: Self−explanatory.

Cannot use color/strict rejection KMC with sectors
: Self−explanatory.

Cannot use create_sites basis with random lattice
: Self−explanatory.

Cannot use diag_style cluster without a lattice defined
: This diagnostic uses the lattice style to dump OpenDx files.

Cannot use dump_one for first snapshot in dump file
: Self−explanatory.

Cannot use random rejection KMC in parallel with no sectors
: Self−explanatory.

Cannot use raster rejection KMC in parallel with no sectors
: Self−explanatory.

Cannot use region INF or EDGE when box does not exist
: Can only define a region with these parameters after a simulation box has been defined.

Choice of sector stop led to no rKMC events
: Self−explanatory.

Color stencil is incommensurate with lattice size
: Since coloring induces a pattern of colors, this pattern must fit an integer number of times into a periodic lattice.

Could not find dump ID in dump_modify command
: Self−explanatory.

Could not find dump ID in dump_one command
: Self−explanatory.

Could not find dump ID in undump command
: Self−explanatory.

Create_box command before app_style set
: Self−explanatory.

Create_box region ID does not exist
: Self−explanatory.

Create_box region must be of type inside
: Self−explanatory.

Create_sites command before app_style set
: Self−explanatory.

Create_sites command before simulation box is defined
: Self−explanatory.

Create_sites region ID does not exist
: Self−explanatory.

Creating a quantity application does not support
: The application defines what variables it supports. You cannot set a variable with the create_sites command for a variable that isn't supported.

Diag dump_style does not work if ncluster > 2^31
: Self−explanatory.

Diag dump_style incompatible with lattice style
: Not all lattice styles can be output as OpenDx files.

Diag propensity requires KMC solve be performed
: Only KMC solvers compute a propensity for sites and the system.

Diag style cluster dump file name too long
: Self−explanatory.

Diag style incompatible with app style
: The lattice styles of the diagnostic and the on−lattice application must match.

Diag cluster dvalue in neighboring clusters do not match
: Internal SPPARKS error.

Diag cluster ivalue in neighboring clusters do not match
: Internal SPPARKS error.

Diag_style command before app_style set
: Self−explanatory.

Diag_style diffusion requires app_style diffusion
: Self−explanatory.

Did not assign all sites correctly
: One or more sites in the read_sites file were not assigned to a processor correctly.

Did not create correct number of sites
: One or more created sites were not assigned to a processor correctly.

Dimension command after lattice is defined
: Self−explanatory.

Dimension command after simulation box is defined
: Self−explanatory.

Divide by 0 in variable formula
: Self−explanatory.

Dump ID already exists
: Self−explanatory.

Dump command before app_style set
: Self−explanatory.

Dump command can only be used for spatial applications
: Self−explanatory.

Dump requires propensity but no KMC solve performed
: Only KMC solvers compute propensity for sites.

Dump_modify command before app_style set
: Self−explanatory.

Dump_one command before app_style set
: Self−explanatory.

Dumping a quantity application does not support
: The application defines what variables it supports. You cannot output a variable in a dump that isn't supported.

Failed to allocate %ld bytes for array %s
: Your SPPARKS simulation has run out of memory. You need to run a smaller simulation or on more processors.

Failed to reallocate %ld bytes for array %s
: Your SPPARKS simulation has run out of memory. You need to run a smaller simulation or on more processors.

Ghost connection was not found
: Internal SPPARKS error. Should not occur.

Ghost site was not found
: Internal SPPARKS error. Should not occur.

Illegal ... command
: Self−explanatory. Check the input script syntax and compare to the documentation for the command. You can use −echo screen as a command−line option when running SPPARKS to see the offending line.

Incorrect args for pair coefficients
: Self−explanatory.

Incorrect lattice neighbor count
: Internal SPPARKS error.

Incorrect site format in data file
: Self−explanatory.

Incorrect value format in data file
: Self−explanatory.

Input line too long after variable substitution
: This is a hard (very large) limit defined in the input.cpp file.

Input line too long: %s
: This is a hard (very large) limit defined in the input.cpp file.

Invalid command−line argument
: One or more command−line arguments is invalid. Check the syntax of the command you are using to launch SPPARKS.

Invalid dump_modify threshold operator
: Self−explanatory.

Invalid event count for app_style test/group
: Number of events must be > 0.

Invalid keyword in dump command
: Self−explanatory.

Invalid math function in variable formula
: The math function is not recognized.

Invalid number of sectors
: Self−explanatory.

Invalid pair style
: Self−explanatory.

Invalid probability bounds for app_style test/group
: Self−explanatory.

Invalid probability bounds for solve_style group
: Self−explanatory.

Invalid probability delta for app_style test/group
: Self−explanatory.

Invalid region style
: Self−explanatory.

Invalid site ID in Sites section of data file
: Self−explanatory.

Invalid syntax in variable formula
: Self−explanatory.

Invalid variable evaluation in variable formula
: A variable used in a formula could not be evaluated.

Invalid variable in next command
: Self−explanatory.

Invalid variable name in variable formula
: Variable name is not recognized.

Invalid variable name
: Variable name used in an input script line is invalid.

Invalid variable style with next command
: Variable styles *equal* and *world* cannot be used in a next command.

Invalid volume setting
: Volume must be set to value > 0.

KMC events are not implemented in app
: Not every application supports KMC solvers.

Label wasn't found in input script
: Self−explanatory.

Lattice command before app_style set
: Self−explanatory.

Lattice style does not match dimension
: Self−explanatory.

Log of zero/negative in variable formula
: Self−explanatory.

Mask logic not implemented in app
: Not every application supports masking.

Mismatch in counting for dbufclust
: Self−explanatory.

Must read Sites before Neighbors
: Self−explanatory.

Must read Sites before Values
: Self−explanatory.

Must use −in switch with multiple partitions
: A multi−partition simulation cannot read the input script from stdin. The −in command−line option must be used to specify a file.

Must use create_sites box for on−lattice applications
: Self−explanatory.

Must use value option before basis option in create_sites command
: Self−explanatory.

No reactions defined for chemistry app
: Use the add_reaction command to specify one or more reactions.

No solver class defined
: Self−explanatory.

Number of sites does not match existing sites
: File being read by read_sites command is not consistent with sites already defined.

One or more sites have invalid values
: The application only allows sites to be initialized with specific values.

PBC remap of site failed
: Internal SPPARKS error.

Pair_coeff command before app_style set
: Self−explanatory.

Pair_coeff command before pair_style is defined
: Self−explanatory.

Pair_style command before app_style set
: Self−explanatory.

Power by 0 in variable formula
: Self−explanatory.

Processor partitions are inconsistent
: The total number of processors in all partitions must match the number of processors LAMMPS is running on.

Processors command after simulation box is defined
: Self−explanatory.

Random lattice has no connectivity
: The cutoff distance is likely too short.

Reaction ID %s already exists
: Cannot re−define a reaction.

Reaction cannot have more than MAX_PRODUCT products
: Self−explanatory.

Reaction has no numeric rate
: Self−explanatory.

Reaction must have 0,1,2 reactants
: Self−explanatory.

Read_sites command before app_style set
: Self−explanatory.

Read_sites simluation box different that current box
: Self−explanatory.

Region command before app_style set
: Self−explanatory.

Region intersect region ID does not exist
: Self−explanatory.

Region union region ID does not exist
: Self−explanatory.

Rejection events are not implemented in app
: Self−explanatory.

Reset_time command before app_style set
: Self−explanatory.

Reuse of region ID
: Self−explanatory.

Run command before app_style set
: Self−explanatory.

Run upto value is before current time
: Self−explanatory.

Seed command has not been used
: The seed command must be used if another command requires random numbers.

Set command before sites exist
: Self−explanatory.

Set command region ID does not exist
: Self−explanatory.

Set if test on quantity application does not support
: The application defines what variables it supports. You cannot do an if test with the set command on a variable that isn't supported.

Setting a quantity application does not support
: The application defines what variables it supports. You cannot set a variable with the set command on a variable that isn't supported.

Simulation box is not multiple of current lattice settings
: This likely occurred because the lattice was re−defined after the simulation box was created.

Site not in my bin domain
: Internal SPPARKS error.

Site−site interaction was not found
: Internal SPPARKS error.

Solve_style command before app_style set
: Self−explanatory.

Species ID %s already exists
Self−explanatory.
Species ID %s does not exist
Self−explanatory.
Sqrt of negative in variable formula
Self−explanatory.
Stats command before app_style set
Self−explanatory.
Substitution for illegal variable
Self−explanatory.
Threshold for a quantity application does not support
The application defines what variables it supports. You cannot do a threshold test with the dump command on a variable that isn't supported.
Too many neighbors per site
Internal SPPARKS error.
Unbalanced quotes in input line
No matching end double quote was found following a leading double quote.
Undump command before app_style set
Self−explanatory.
Unexpected end of data file
Self−explanatory.
Universe/uloop variable count < # of partitions
A universe or uloop style variable must specify a number of values >= to the number of processor partitions.
Unknown command: %s
The command is not known to SPPARKS. Check the input script.
Unknown identifier in data file: %s
Self−explanatory.
Unknown species in reaction command
Self−explanatory.
Unrecognized command
The command is assumed to be application specific, but is not known to SPPARKS. Check the input script.
Use of region with undefined lattice
The lattice command must be used before defining a geometric region.
Variable name must be alphanumeric or underscore characters
Self−explanatory.
World variable count doesn't match # of partitions
A world−style variable must specify a number of values equal to the number of processor partitions.

Warnings:

add_reaction command

Syntax:

```
add_reaction reactant1 reactant2 rate product1 product2 ...
```

- reactant1,reactant2 = 0, 1, or 2 reactant species
- rate = reaction rate (see units below)
- product1, product2 = 0, 1, or more product species

Examples:

```
add_reaction A B 1.0e10 C
add_reaction 1.0 d
add_reaction b2 1.0e−10 c3 d4 e3
```

Description:

This command defines a chemical reaction for use in the app_style chemistry application.

Each reaction has 0, 1, or 2 reactants. It also has 0, 1, or more products. The reactants and products are specified by species ID strings, as defined by the add_species command.

The units of the specified rate constant depend on how many reactants participate in the reaction:

- 0 reactants = rate is molarity/sec
- 1 reactant = rate is 1/sec
- 2 reactants = rate is 1/molarity−sec

Thus the first reaction listed above represents an A and B molecule binding to form a complex C at a rate of 1.0e10 per molarity per second. I.e. A + B −> C.

Restrictions:

This command can only be used as part of the app_style chemistry application.

Related commands:

app_style chemistry, add_species

Default: none

add_species command

Syntax:

```
add_species name1 name2 ...
```

- name1,name2 = ID strings for different species

Examples:

```
add_species kinase
add_species NFkB kinase2 NFkB−IKK
```

Description:

This command defines the names of one or more chemical species for use in the app_style chemistry application.

Each ID string can be any sequence of non−whitespace characters (alphanumeric, dash, underscore, etc).

Restrictions:

This command can only be used as part of the app_style chemistry application.

Related commands:

app_style chemistry, add_reaction, count

Default: none

app_style chemistry command

Syntax:

```
app_style chemistry
```

- chemistry = application style name

Examples:

```
app_style chemistry
```

Description:

This is a general application which evolves a set of coupled chemical reactions stochastically, producing a time trace of species concentrations. Chemical species are treated as counts of individual molecules reacting within a reaction volume in a well−mixed fashion. Individual reactions are chosen via the direct method variant of the Stochastic Simulation Algorithm (SSA) of (Gillespie).

A prototypical example is to use this model to simulate the execution of a protein signaling network in a biological cell.

This application can only be evolved using a kinetic Monte Carlo (KMC) algorithm. You must thus define a KMC solver to be used with the application via the solve_style command

---

The following additional commands are defined by this application:

add_reaction	define a chemical reaction
add_species	define a chemical species
count	specify molecular count of a species
stats	output of system info
volume	specify volume of the chemical reactor

Restrictions: none

Related commands: none

Default: none

(Gillepsie) Gillespie, J Chem Phys, 22, 403−434 (1976); Gillespie, J Phys Chem, 81, 2340−2361 (1977).

app_style diffusion command

Syntax:

```
app_style diffusion estyle dstyle args gstyle args keyword values ...
```

- diffusion = application style name
- estyle = *off* or *linear* or *nonlinear*
- dstyle = *hop* or *schwoebel*

```
hop args = none
  schwoebel args = Nmax Nmin
    Nmax = max # of neighbors the initial Schwoebel site can have
    Nmin = min # of neighbors the final Schwoebel site can have
```

- gstyle = *none* or *surface* or *void* or *pore*

```
none args = none, use of input file option is assumed
  surface args = level
    level = height of initial surface in y (2d) or z (3d)
  void args = fraction
    fraction = random fraction of sites that are initially occupied
  pore args = xc yc zc diameter thickness
    xc,yc,zc = coordinates of center point of pore
    diameter = xy diameter of cylindrical pore aligned along z axis
    thickness = z thickness of thin film which the pore spans
```

- see the app_style command for additional keywords that can be appended

Examples:

```
app_style diffusion linear hop void 0.2 lattice sq/4n 1.0 20 20
app_style diffusion nonlinear schwoebel 5 2 void 0.2 lattice fcc 1.0 50 50
app_style diffusion off hop pore 10 10 10 5 15 &                    lattice fcc 1.0 20 20 20
```

Description:

This is an on−lattice application which performs diffusive hops on a lattice whose sites are partially occupied and partially unoccupied (vacancies). It can be used to model surface diffusion or bulk diffusion on 2d or 3d lattices. It is equivalent to a 2−state Ising model performing Kawasaki dynamics where neighboring sites exchange their spins as the model evolves.

The *estyle* setting determines how energy is used in computing the probability of hop events, which is related to the Hamiltonian for the system.

The Hamiltonian representing the energy of an occupied site I for the *off* style is 0, which simply means energy is not used in determining the hop probabilities. Instead, see the barrier command.

The Hamiltonian representing the energy of an occupied site I for the *linear* style is as follows:

```
Hi = Sum_j delta_ij
```

where Sum_j is a sum over all the neighbor sites of site I and delta_ij is 0 if site J is occupied and 1 if site J is vacant. The Hi for a vacant site is 0.

The Hamiltonian representing the energy of an occupied site I for the *nonlinear* style is as follows:

```
Hi = Eng(Sum_j delta_ij)
```

where Sum_j is the sum over all its neighbor sites and delta_ij now 1 if site J is occupied and 0 otherwise. Thus the summation computes the coordination number of site I. Note that this definition of delta is the opposite of how it is defined for estyle *linear*. The function Eng() is a tabulated function with values specified via the ecoord command. This effectively allows the energy to be a non−linear function of coordination number. As before the Hi for a vacant site is 0.

For all these *estyle* settings, the energy of the entire system is the sum of Hi over all sites.

The *dstyle* setting determines what kind of diffusive hops are modeled. For *hop*, only simple nearest−neighbor hops occur where an atom hops to a neighboring vacant site. For *schwoebel*, Schwoebel hops can also occur, which are defined in the following way. An atom I can hop to a 2nd neighbor vacant site K if there are two intermediate 1st neighbor sites J1 and J2, where J1 is vacant and J2 is occupied, and J1 and J2 are neighbors of each other. Additionaly the initial site I can have no more the *Nmax* occupied neighbors (its coordination number), and the destination site K can have no fewer than *Nmin* neighbors.

The *gstyle* setting determines how the lattice geometry is initialized.

For gstyle *none*, no explicit geometric initialization is performed, rather it is assumed the *lattice file* option is used. See the app_style command for details.

For gstyle *surface*, a flat surface is initialized. All sites below *level* in 2d or 3d will be occupied. Sites above the surface will be vacant.

For gstyle *void*, the specified *fraction* of sites will be occupied, the remained will be vacant. Which sites are occupied versus vacant is determined randomly. Thus this sets up a bulk system with (1−fraction) percent void fraction.

For gstyle *pore*, a thin film of *thickness* is setup with a cylindrical pore of *diameter*, centered at (xc,yc,zc). The axis of the pore is along y for a 2d system and along z for a 3d system.

The deposition command can be used with this application to add atoms to the system in competition with hop events.

The barrier command can be used with this application to add an energy barrier to the model for nearest−neighbor hop and Schwoebel hop events, as discussed below.

The ecoord command can be used with the *nonlinear* version of this application to set tabulated values for the Hamiltonian Eng() function as described above.

Note that estyle *nonlinear* should give the same answer as estyle *linear* if the tabulated function specified by the ecoord command is specified as E_0 = N, E_1 = N−1, ... E_N−1 = 1, E_N = 0, where N = the number of neighbors of each lattice site, i.e. the maximum coordination number. In this scenario, the energy is effectively a linear function of coordination number.

This application is a general lattice application; see the app_style commmand for further discussion. The lattice must be specified by the appended *lattice* keword with its associated values, as discussed on the doc page for the app_style command.

This application performs Kawasaki dynamics, in that the "spins" on two neighboring sites are swapped, where spin can be thought of as a flag representing occupied or vacant. Equivalently, an atom hops from an occupied site to a vacancy site.

As explained on this page, this application can be evolved by either a kinetic Monte Carlo (KMC) or rejection KMC (rKMC) algorithm. You must thus define a KMC solver or sweeping method to be used with the application via the solve_style or sweep commands. The *linear* estyle supports both KMC and rKMC options. The other estyles only support KMC options. If the deposition command is used, then only KMC options are supported.

For solution by a KMC algorithm, the possible events an occupied site can perform are swaps with vacant neighbor sites. The probability of each such event depends on several variables: the *estyle* setting, whether the barrier command is used, whether the hop is downhill or uphill in energy, and whether the temperature is 0.0 or finite. The following table gives the hop probability for each possible combination of these variables.

Energy	Barrier	Direction	Temperature	Probability
no	no	N/A	either	1
no	yes	N/A	0.0	0
no	yes	N/A	finite	exp(−Q/kT)
yes	no	down	either	1
yes	no	up	0.0	0
yes	no	up	finite	exp(−dE/kT)
yes	yes	down	0.0	0
yes	yes	down	finite	exp(−Q/kT)
yes	yes	up	0.0	0
yes	yes	up	finite	exp((−dE−Q)/kT)

If *estyle* is set to *off*, then energy is "no" in the table. Any other *estyle* setting is energy = "yes". Barrier is "no" in the table if the "barrier" command is not used, else it is "yes" in the table. The direction of energy change (downhill versus uphill) is only relevant if energy is "yes", else it is N/A. The "either" entry for temperature means 0.0 or finite.

The value dE = Efinal − Einitial refers to the energy change in the system due to the hop. For estyle *linear* this can be computed from just the sites I,J. For estyle *nonlinear* the energy of the neighbors of both sites I,J must also be computed.

For solution by a Metropolis algorithm, the hop event is performed or not if the probability in the table is 1 or 0. For intermediate values, a uniform random number R between 0 and 1 is generated and the hop event is accepted if R < probability in the table.

The following additional commands are defined by these applications. The ecoord command can only be used with the *nonlinear* energy style.

barrier	define energy barriers for hop events
deposition	define deposition events
dump	output of lattice snapshots
dump_one	one snapshot of lattice state
ecoord	specify site energy as a function of coordination number
stats	output of system info
temperature	set Monte Carlo temperature

Restrictions: none

Related commands:

app_style ising

Default: none

app_style ising command

app_style ising/single command

Syntax:

```
app_style style keyword values ...
```

- style = *ising* or *ising/single*
- see the app_style command for additional keywords that can be appended to the *ising* style

Examples:

```
app_style ising lattice sq/4n 1.0 50 50
app_style ising/single lattice sq/4n 1.0 50 50
```

Description:

These are on−lattice applications which evolve a 2−state Ising model, where each lattice site has a spin of 1 or 2. Sites flip their spin as the model evolves.

The Hamiltonian representing the energy of site I is as follows:

```
Hi = Sum_j delta_ij
```

where Sum_j is a sum over all the neighbor sites of site I and delta_ij is 0 if the spin of sites I and J are the same and 1 if they are different. The energy of the entire system is the sum of Hi over all sites.

This application performs Glauber dynamics, meaning the spin is flipped on a single site. See app_style diffusion for an Ising model which performs Kawasaki dynamics, meaning the spins on two neighboring sites are swapped.

As explained on this page, this application can be evolved by either a kinetic Monte Carlo (KMC) or rejection KMC (rKMC) algorithm. You must thus define a KMC solver or sweeping method to be used with the application via the solve_style or sweep commands.

For solution by a KMC algorithm, a site event is a spin flip and its probability is min[1,exp(−dE/kT)], where dE = Efinal − Einitial using the Hamiltonian defined above for the energy of the site, and T is the temperature of the system defined by the temperature command (which includes the Boltzmann constant k implicitly).

For solution by a rKMC algorithm, the *ising* and *ising/single* styles use a different rejection−based algorithm. For the *ising* style, the spin is set randomly to 1 or 2. For the *ising/single* style, the spin is flipped to its opposite value. In either case, dE = Efinal − Einitial is calculated, as is a uniform random number R between 0 and 1. The new state is accepted if R < min[1,exp(−dE/kT)].

The following additional commands are defined by this application:

dump	output of lattice snapshots
dump_one	one snapshot of lattice state
stats	output of system info

temperature	set Monte Carlo temperature

Restrictions: none

Related commands:

app_style potts

Default: none

app_style membrane command

Syntax:

```
app_style membrane w01 w11 mu keyword values ...
```

- membrane = style name of this application
- w01 = sovent−protein interaction energy (typically 1.25)
- w11 = sovent−solvent interaction energy (typically 1.0)
- mu = chemical potential to insert a solvent (typically −2.0)
- see the app_style command for additional keywords that can be appended to the *membrane* style

Examples:

```
app_style membrane 1.25 1.0 −3.0 lattice tri 1.0 100 50
```

Description:

This is an on−lattice application which evolves a membrane model, where each lattice site is in one of 3 states: lipid, water, or protein. Sites flip their state as the model evolves. See the paper of (Sarkisov) for a description of the model and its applications to porous media. Here it is used to model the state of a lipid membrane around embedded proteins, such as one enclosing a biological cell.

In the model, protein sites are defined by the inclusion command and never change. The remaining sites are initially lipid and can flip between solvent and lipid as the model evolves. Typically, water will coat the surface of the proteins and create a pore in between multiple proteins if they are close enough together.

The Hamiltonian represeting the energy of site I is as follows:

```
H = − mu x_i − Sum_j (w11 a_ij + w01 b_ij)
```

where Sum_j is a sum over all the neighbor sites of site I, x_i = 1 if site I is solvent and 0 otherwise, a_ij = 1 if both the I,J sites are solvent and 0 otherwise, b_ij = 1 if one of the I,J sites is solvent and the other is protein and 0 otherwise. Mu and w11 and w01 are user inputs. As discussed in the paper, this is essentially a lattice gas grand−canonical Monte Carlo model, which is isomorphic to an Ising model. The mu term is a penalty for inserting solvent which prevents the system from becoming all solvent, which the 2nd term would prefer.

As explained on this page, this application can be evolved by either a kinetic Monte Carlo (KMC) or rejection KMC (rKMC) algorithm. You must thus define a KMC solver or sweeping method to be used with the application via the solve_style or sweep commands.

For solution by a KMC algorithm, a site event is a spin flip from a lipid to fluid state or vice versa. The probability of the event is min[1,exp(−dE/kT)], where dE = Efinal − Einitial using the Hamiltonian defined above for the energy of the site, and T is the temperature of the system defined by the temperature command (which includes the Boltzmann constant k implicitly).

For solution by a Metropolis algorithm, the site is set randomly to fluid or lipid, unless it is a protein site in which case it is skipped altogether. The energy change dE = Efinal − Einitial is calculated, as is a uniform random number R between 0 and 1. The new state is accepted if R < min[1,exp(−dE/kT)], else it is rejected.

The following additional commands are defined by these applications:

dump	output of lattice snapshots
dump_one	one snapshot of lattice state
inclusion	specify which sites are proteins
stats	output of system info
temperature	set Monte Carlo temperature

Restrictions: none

Related commands: none

Default: none

(Sarkisov) Sarkisov and Monson, Phys Rev E, 65, 011202 (2001).

app_style potts command

app_style potts/neigh command

app_style potts/neighonly command

Syntax:

```
app_style style Q keyword values ...
```

- style = *potts* or *potts/neigh* or *potts/neighonly*
- Q = number of spin states
- see the app_style command for additional keywords that can be appended

Examples:

```
app_style potts 100 lattice sq/4n 1.0 50 50
app_style potts/neigh 20 lattice sq/4n 1.0 50 50
```

Description:

These are on−lattice applications which evolve a Q−state Ising model or Potts model, where each lattice site has a spin value from 1 to Q. Sites flip their spin as the model evolves.

The Hamiltonian representing the energy of site I is as follows:

```
Hi = Sum_j delta_ij
```

where Sum_j is a sum over all the neighbor sites of site I and delta_ij is 0 if the spin of sites I and J are the same and 1 if they are different. The energy of the entire system is the sum of Hi over all sites.

These applications perform Glauber dynamics, meaning the spin is flipped on a single site. See app_style diffusion for an Ising model which performs Kawasaki dynamics, meaning the spins on two neighboring sites are swapped.

As explained on this page, these applications can be evolved by either a kinetic Monte Carlo (KMC) or rejection KMC (rKMC) algorithm. You must thus define a KMC solver or sweeping method to be used with the application via the solve_style or sweep commands.

For solution by a KMC algorithm, a site event is a spin flip and its probability is min[1,exp(−dE/kT)], where dE = Efinal − Einitial using the Hamiltonian defined above for the energy of the site, and T is the temperature of the system defined by the temperature command (which includes the Boltzmann constant k implicitly). The KMC algorithm does not allow spin flips known as "wild" flips, even at finite temperature. These are flips to values that are not equal to any neighbor site value.

For solution by a rKMC algorithm, the various styles use different rejection−based algorithms. For the *potts* style, a random spin from 1 to Q is chosen. For the *potts/neigh* style, a spin is chosen randomly from the values held by neighbor sites and a null−bin of a size which extends the possible events up to the maximum number of neighbors. For example, imagine a site has 12 neighbors and the 12 sites have 4 different spin values. Then each of the 4 neighbor spin values will be chosen with 1/12 probability and the null bin will be chosen with 8/12

probability. For the *potts/neighonly* style, the null bin is discarded, so in this case each of the 4 spin values will be chosen with 1/4 probability. In all the cases, dE = Efinal − Einitial is calculated, as is a uniform random number R between 0 and 1. The new state is accepted if R < min[1,exp(−dE/kT)], else it is rejected.

The rKMC algorithm for the *potts* style does allow spin flips known as "wild" flips. These are flips to values that are not equal to any neighbor site value. At temperature 0.0 these are effectively disallowed, since they will increase the energy of the system (except in the uninteresting case when the site already has a spin value not equal to any neighbor values), but at finite temperature they will have a non−zero probability of occurring.

The following additional commands are defined by these applications:

dump	output of lattice snapshots
dump_one	one snapshot of lattice state
stats	output of system info
temperature	set Monte Carlo temperature

Restrictions: none

Related commands:

app_style ising

Default: none

app_style potts/pin command

Syntax:

```
app_style potts/pin Q keyword values ...
```

- potts/pin = application style name
- Q = number of spin states
- see the app_style command for additional keywords that can be appended

Examples:

```
app_style potts/pin 100 lattice tri 1.0 50 50
```

Description:

This is an on−lattice application which evolves a Q−state Potts model in the presence of pinning sites, which are sites tagged with a spin value of Q+1 which do not change. Their effect is typically to pin or inhibit grain growth in various ways.

The Hamiltonian representing the energy of site I is as follows:

```
Hi = Sum_j delta_ij
```

where Sum_j is a sum over all the neighbor sites of site I and delta_ij is 0 if the spin of sites I and J are the same and 1 if they are different. The energy of the entire system is the sum of Hi over all sites.

These applications perform Glauber dynamics, meaning the spin is flipped on a single site. See app_style diffusion for an Ising model which performs Kawasaki dynamics, meaning the spins on two neighboring sites are swapped.

As explained on this page, this application can be evolved by either a kinetic Monte Carlo (KMC) or rejection KMC (rKMC) algorithm. You must thus define a KMC solver or sweeping method to be used with the application via the solve_style or sweep commands.

For solution by a KMC algorithm, a site event is a spin flip and its probability is min[1,exp(−dE/kT)], where dE = Efinal − Einitial using the Hamiltonian defined above for the energy of the site, and T is the temperature of the system defined by the temperature command (which includes the Boltzmann constant k implicitly). The KMC algorithm does not allow spin flips known as "wild" flips, even at finite temperature. These are flips to values that are not equal to any neighbor site value. The KMC algorithm also does not allow spin flips to a pinned site value.

For solution by a rKMC algorithm, a random spin from 1 to Q is chosen. Note that this does not allow a spin flip to a pinned site value, since those sites are set to Q+1. When the flip is attempted dE = Efinal − Einitial is calculated, as is a uniform random number R between 0 and 1. The new state is accepted if R < min[1,exp(−dE/kT)], else it is rejected.

The rKMC algorithm for the *potts* style does allow spin flips known as "wild" flips. These are flips to values that are not equal to any neighbor site value. At temperature 0.0 these are effectively disallowed, since they will increase the energy of the system (except in the uninteresting case when the site already has a spin value not equal to any neighbor values), but at finite temperature they will have a non−zero probability of occurring.

The following additional commands are defined by this application:

dump	output of lattice snapshots
dump_one	one snapshot of lattice state
pin	create a set of pinned sites
stats	output of system info
temperature	set Monte Carlo temperature

Restrictions: none

Related commands:

app_style potts

Default: none

app_style relax command

Syntax:

```
app_style relax delta keyword values ...
```

- relax = style name of this application
- delta = maximum displacement distance of a particle (distance units)
- see the app_style command for additional keywords that can be appended

Examples:

```
app_style relax 0.5 lattice sq/4n 1.0 50 50
```

Description:

This is an off−lattice application which treats sites as particles which interact through a pair potential and whose collective energy is relaxed via Metropolis Monte Carlo translational moves.

The energy of a particle I is as follows:

```
Ei = Sum_j phi(Rij)
```

where Sum_j is a sum over all the neighbor of pariticle I within some cutoff distance, phi() is the potential energy function defined by the pair_style command, and Rij is the distance between particles I and J. The energy of the entire system is the sum of Ei over all particles. The pair_style command also defines the cutoff distance.

As explained on this page, this application is evolved by a Metroplis Monte Carlo (MMC) algorithm. You must thus define a sweeping method to be used with the application via the sweep command.

For solution by the MMC algorithm, once a particle is chosen, a translational move of the particle is made, by choosing a random location within a sphere of radius *delta* surrounding the particle. The energy of the particle before and after the move is calculated, to give dE = Efinal − Einitial. The move is accepted if R < min[1,exp(−dE/kT)], else it is rejected, where R is a uniform random number R between 0 and 1.

The following additional commands are defined by this application:

dump	output of particle snapshots
dump_one	one snapshot of particle coords
stats	output of system info
temperature	set Monte Carlo temperature

Restrictions: none

Related commands:

Default: none

app_style command

Syntax:

```
app_style style args
```

- style = one of a list of possible style names (see below)
- args = arguments specific to an application, see application doc page for details

Examples:

```
app_style diffusion ...
app_style ising ...
app_style potts ...
app_style relax ...
app_style chemistry ...
app_style test/group ...
```

Description:

This command defines what model or application SPPARKS will run. There are 3 kinds of applications: on−lattice, off−lattice, and general.

On−lattice applications define a set of static sites in space on which events occur. The sites can represent a crystalline lattice, or be more disordered. The key point is that they are immobile and that each site's neighborhood of nearby sites can be specified. Here is the list of on−lattice applications SPPARKS currently includes:

- diffusion = vacancy exchange diffusion model
- ising = Ising model
- ising/single = variant Ising model
- membrane = membrane model of lipid,water,protein
- potts = Potts model for grain growth
- potts/neigh = variant Potts model
- potts/neighonly = variant Potts model
- potts/pin = Potts model with pinning sites

Off−lattice applications define a set of mobile sites in space on which events occur. The sites typically represent particles. Each site's neighborhood of nearby sites is defined by a cutoff distance. Here is the list of off−lattice applications SPPARKS currently includes.

- relax = Metropolis Monte Carlo relaxation

General applications require no spatial information. Events are defined by the application, as well as the influence of each event on others. Here is the list of general applications SPPARKS currently includes.

- chemistry = biochemical reaction networks
- test/group = artificial chemical networks that test solve_style

The general applications in SPPARKS can only be evolved via a kinetic Monte Carlo (KMC) solver, specified by the solve_style command. On−lattice and off−lattice applications can be evolved by either a KMC solver or a

rejection kinetic Monte Carlo (rKMC) method or a Metropolis (MMC) method. The rKMC and MMC methods are specified by the sweep command. Not all on− and off−lattice applications support each option.

KMC models are sometimes called rejection−free KMC or the N−fold way or the Gillespie algorithm in the MC literature. The application defines a list of "events" and associated rates for each event. The solver chooses the next event, and the application updates the system accordingly. This includes updating of the time, which is done accurately since rates are defined for each event. For general applications the definition of an "event" is arbitrary. For on−lattice application zero or more possible events are typically defined for each site.

rKMC models are sometimes called null−event KMC or null−event MC. Sites are chosen via some method (see the sweep command), and an event on that site is then selected which is accepted or rejected. Again, the application defines the "events" for each site and associated rates which influence the acceptance or rejection. It also defines the null event which is essentially part of the rejection probability.

For KMC and rKMC models, a time is associated with each event (including the null event) by rates that the user defines. Thus event selection induces a time−accurate simulation. The MMC method is similar to the rKMC method, except that it is not time−accurate. It selects an event to perform and accepts or rejects it, typically based on an energy change in the system. There is no rate associated with the event, and no requirement that events be chosen with relative probabilities corresponding to their rates. The Metropolis method tends to evolve the system towards a low energy state. As with the rKMC method, the sweep command is used to determine how sites are selected.

For all three methods (KMC, rKMC, MMC) the rules for how events are defined and are accepted or rejected are discussed in the doc pages for the individual applications.

This table lists the different kinds of solvers and sweeping options that can be used for on− and off−lattice applications in SPPARKS. Serial and parallel refer to running on one or many processors. Sector vs no−sector is what is set by the sector command. The rKMC options are set by the sweep command. The MMC options are the same as for rKMC.

method	serial/no−sectors	serial/sectors	parallel/no−sectors	parallel/sectors
exact KMC	yes	yes	no	yes
rKMC random	yes	yes	no	yes
rKMC raster	yes	yes	no	yes
rKMC color	yes	no	yes	no

Note that the choice of *color* can also be *color/strict* and that masking can also be turned on for rKMC algorithms via the sweep command if the application supports it. Off−lattice applications do not support the *color* or *masking* options.

Restrictions: none

Related commands: none

Default: none

app_style test/group command

Syntax:

```
app_style test/group N Nmax pmax pmin delta keyword value
```

- test/group = application style name
- N = # of events to choose from
- Mmax = max number of dependencies for each event
- pmax = max probability
- pmin = min probability
- delta = percentage adjustment factor for dependent probabilities
- zero or more keyword/value pairs may be appended
- keyword = *lomem*

  lomem value = *yes* or *no*

Examples:

```
app_style test/group 10000 30 1.0 1.0e-6 5
app_style test/group 10000 30 1.0 1.0e-9 10 lomem yes
```

Description:

This is a general application which creates and evolves an artificial network of coupled events to test the performance and scalability of various kinetic Monte Carlo solvers. See the paper by (Slepoy) for additional details on how it has been used.

The set of coupled events can be thought of as a reaction network with N different chemical rate equations or events to choose from. Each equation is coupled to M randomly chosen other equations, where M is a uniform random number from 1 to Mmax. In a chemical reaction sense it is as if an executed reaction creates M product molecules, each of which is a reactant in another reaction, affecting its probability of occurrence.

Initially, the maximum and minimum probability for each event is an exponentially distributed random value between *pmax* and *pmin*. If solve_style group is used, these values should be the same as the *pmax* and *pmin* used as parameters in that command. Pmin must be greater than 0.0.

As events are executed, the artificial network updates the probabilities of dependent reactions directly by adjusting their probability by a uniform random number betwee −delta and +delta. Since delta is specified as a percentatge, this means pold * (1 − delta/100) <= pnew <= pold * (1 + delta/100). Delta must be less than 100.

If the *lomem* keyword is set to *no*, then the random connectivity of the network is generated beforehand and stored. This is faster when events are executed but limits the size of problem that will fit in memory. If *lomem* is set to *yes*, then the connectivity is generated on the fly, as each event is executed.

This application can only be evolved using a kinetic Monte Carlo (KMC) algorithm. You must thus define a KMC solver to be used with the application via the solve_style command

When the run command is used with this application it sets the number of events to perform, not the time for the run. E.g.

```
run 10000
```

means to perform 10000 events, not to run for 10000 seconds.

The following additional command is defined by this application:

stats	output of system info

Restrictions: none

Related commands:

solve_style group

Default:

The default value is lomem = no.

(Slepoy) Slepoy, Thompson, Plimpton, J Chem Phys, 128, 205101 (2008).

barrier command

Syntax:

```
barrier dstyle Q
barrier dstyle delta Q
barrier dstyle I J Q
```

- dstyle = *hop* or *schwoebel*
- Q = barrier height (energy units)
- delta = difference in coordination number of 2 participating sites
- I,J = coordination numbers of 2 participating sites

Examples:

```
barrier hop 0.25
barrier schwoebel 1 0.3
barrier hop −1 0.35
barrier hop 3 4 0.2
barrier schwoebel * * 0.1
barrier hop 2*5 3* 0.1
```

Description:

This command sets the energy barrier for a diffusive hop of an atom from an occupied site to a nearby vacant site. See the app_style diffusion command for how the barrier is used in conjunction with the energy change of the system due to the hop to calculate a probability for the hop to occur.

Barriers can be assigned to two kinds of diffusive hops. The first is a hop to a nearest−neighbor vacancy, which is specified by setting *dstyle* to *hop*. The second is a Schwoebel hop to a 2nd nearest−neighbor vacancy, which is specified by setting *dstyle* to *schwoebel*. The latter is only allowed if the app_style diffusion command also used *schwoebel* for its dstyle setting.

Barriers are assigned based on two coordination numbers, for the initial site of the hopping atom and its final site. In both cases the coordination count does not include the hopping atom itself. Thus typically (Nmax+1)*(Nmax+1) values should be specified by using this command one or more times, which can be thought of as an (I,J) matrix entries where both I and J vary from 0 to Nmax inclusive, when Nmax is the number of neighbor sites for each lattice site. There is one such matrix for nearest−neighbor diffusive hops and one for Schwoebel hops. Also note that it is permissible to have Qij != Qji to set forward/reverse rates, particularly if the model does not use energies, but only barriers.

If only one argument Q is specified, then all matrix values are set to Q. If the Q value = 0.0, this effectively turns off barriers in the model.

If two arguments delta and Q are specified, then all matrix elements where delta = J−I are set to Q.

If three arguments I and J and Q are specified, then the (I,J) element is set to Q. In this case, the I.J indices can each be specified in one of two ways. An explicit numeric value can be used, as in the 4th example above. Or a wild−card asterisk can be used to set the energy value for multiple coordination numbers. This takes the form "*" or "*n" or "n*" or "m*n". If Nmax = the number of neighbor sites, then an asterisk with no numeric values means all coordination numbers from 0 to Nmax. A leading asterisk means all coordination numbers from 0 to n (inclusive). A trailing asterisk means all coordination numbers from n to Nmax (inclusive). A middle asterisk

means all coordination numbers from m to n (inclusive).

The *Q* value should be in the energy units defined by the application's Hamiltonian and should be consistent with the units used in any temperature command.

Restrictions:

This command can only be used as part of the app_style diffusion application.

Related commands:

deposition, ecoord

Default:

Energy barriers for all hop events are set to 0, which is effectively no barriers.

clear command

Syntax:

```
clear
```

Examples:

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

Description:

This command deletes all data, restores all settings to their default values, and frees all memory allocated by SPPARKS. Once a clear command has been executed, it is as if SPPARKS 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

count command

Syntax:

```
count species N
```

- species = ID of chemical species
- N = count of molecules of this species

Examples:

```
count kinase 10000
count NFkB-IKK 300
```

Description:

This command sets the molecular count of a chemical species for use in the app_style chemistry application.

The species ID can be any string defined by the add_species command.

Restrictions:

This command can only be used as part of the app_style chemistry application.

Related commands:

app_style chemistry, add_species, add_reaction

Default:

The count of a defined species is 0 unless set via this command.

create_box command

Syntax:

```
create_box region-ID
```

- region−ID = ID of region to use as simulation domain

Examples:

```
create_box mybox
```

Description:

This command creates a simulation box based on the specified region for on−lattice and off−lattice spatial simulations. Thus a region command must first be used to define a geometric domain. SPPARKS encloses the region (block, sphere, etc) with an axis−aligned (orthogonal) box which becomes the simulation domain.

The read_sites command can also be used to define a simulation box.

Restrictions:

The app_style command must be used to define an application before using the create_box command.

Related commands:

create_sites, region, read_sites

Default: none

create_sites command

Syntax:

```
create_sites style arg keyword values ...
```

- style = *box* or *region*

```
  box arg = none
    region arg = region-ID
      region-ID = sites will only be created if contained in the region
```

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

```
  value values = label nvalue
      label = site or iN or dN
      nvalue = specific value to set all created sites to
    basis values = M nvalue
      M = which basis site (see asterisk form below)
      nvalue = specific value to set all created basis sites to
```

Examples:

```
create_sites box
create_sites region surf value site 1
create_sites box value i2 0 basis 1 1 basis 2* 2
```

Description:

This command creates "sites" on a lattice for on−lattice and off−lattice applications. This is an alternative to reading in their coordinates via a read_sites command. A simulation box must already exist, which is typically created via the create_box command. Before using this command, a lattice must also be defined using the lattice command.

In SPPARKS, a "site" is a point in space at which an application, as defined by the app_style command can perform events. For on−lattice applications, the site is static and has a defined set of neighboring sites with which it interacts. For off−lattice applications, a site is like a particle. It moves and has a dynamic neighborhood of nearby particles with which it interacts.

For the *box* style, the create_sites command fills the entire simulation box with sites on the lattice. This is the only option allowed for on−lattice applications.

In this case the simulation box size must be an integer multiple of the lattice constant in each dimension, to insure consitent placement of sites and neighbor interactions across periodic boundaries. SPPARKS is careful to put exactly one site at the boundary (on either side of the box), not zero or two.

NOTE: The *region* style is not yet implemented but will be soon. The following text explains how it will work.

For the *region* style, the geometric volume is filled with sites on the lattice that are both inside the simulation box and also consistent with the region volume. See the region command for details. Note that a region can be specified so that its "volume" is either inside or outside a geometric boundary. Also note that even if the region is the same size as a periodic simulation box (in some dimension), SPPARKS does not implement the same logic as

with the *box* style, to insure exactly one site at the boundary. If this is what you desire, you should either use the *box* style, or tweak the region size to get precisely the sites you want.

Site IDs are assigned to created sites with consecutive values from 1 to N, where N is the total number of sites that fill the simulation box. The numbering is the same, independent of the number of processors.

Depending on the application, each site stores zero of more integer and floating−point values. By default these are set to zero when a site is created by this command. The *value* and *basis* keywords can override the default.

The *value* keyword specifies a per−site value that will be assigned to every site as it is created. The *label* determines which per−site quantity is set. *iN* and *dN* mean the Nth integer or floating−point quantity, with 1 <= N <= Nmax. Nmax is defined by the application. If *label* is specified as *site* it is the same as *i1*. The quantity is set to the specified *nvalue*, which should be either an integer or floating−point numeric value, depending on what kind of per−site quantity is being set.

The *basis* keyword can be used to override the *value* keyword setting for individual basis sites as each unit cell is created. The per−site quantity (e.g. i2) specified by the *value* keyword is set for basis sites *M*. The quantity is set to the specified *nvalue* for the *basis* keyword, instead of the *nvalue* from the *value* keyword. See the lattice command for specifics on how basis atoms and unit cells are defined for each lattice style.

M can be specified in one of two ways. An explicit numeric value can be used, such as 2. A wild−card asterisk can also be used in place of or in conjunction with the M argument to specify multiple basis sites together. This takes the form "*" or "*n" or "n*" or "m*n". If N = the total number of basis sites, then an asterisk with no numeric values means all sites from 1 to N. A leading asterisk means all sites from 1 to n (inclusive). A trailing asterisk means all sites from n to N (inclusive). A middle asterisk means all sites from m to n (inclusive).

Restrictions:

The app_style command must be used to define an application before using the create_box command.

Related commands:

lattice, region, create_box, read_sites

Default: none

deposition command

Syntax:

```
deposition rate dirx diry dirz d0 lo hi
```

- rate = rate of atom deposition (atom/sec units)
- dirx,diry,dirz = vector in direction of incidence
- d0 = capture distance (distance units)
- lo,hi = min/max coordination number of deposition site

Examples:

```
deposition       1.0 0 −1 0 1.0 1 4
deposition       1.0 1 1 −1 1.0 3 10
```

Description:

This commands invokes deposition events in an on−lattice diffusion model, specified by the app_style diffusion command.

Deposition events will compete with diffusive hop events in the diffusion model. Each time a deposition event is selected, a random starting point at the top of the simulation box is selected (top y surface in 2d, top z surface in 3d). The atom trajectory is traced along its incident direction which is specified by (dirx,diry,dirz) and need not be a unit vector. However, diry < 0 and dirz = 0 is required for 2d models. Similarly, dirz < 0 is required for 3d models.

Candidate deposition sites are vacant sites within a perpendicular distance *d0* from the incident trajectory which also have a current coordination number C such that *lo* <= C <= *hi*. Note that *d0* is specified in distance units which will depend on how you setup your lattice via the app_style command. For example, if you specified you lattice constant or box size in Angstroms, then the distance units for this command are Angstroms as well.

If the inicident angle is not vertical, then periodic images of the starting point with associated incident trajectories are considered and the *d0* capture distance is applied to whichever trajectory the candidate site is closest to, in a perpendicular sense. This means x−periodicity in 2d and x− and y−periodicity in 3d.

For the set of candidate sites, the selected deposition site is the one closest to the starting point, measuring a projected distance along the incident direction.

Restrictions:

This command can only be used as part of the app_style diffusion application.

Deposition can currently only be done in serial simulations, not parallel.

Related commands:

ecoord, barrier

Default: none

diag_style cluster command

Syntax:

```
diag_style cluster keyword value keyword value ...
```

- cluster = style name of this diagnostic
- zero or more keyword/value pairs may be appended
- see the diag_style command for additional keywords that can be appended to a diagnostic command and which must appear before these keywords
- keyword = *filename* or *dump*

```
filename value = name
    name = name of file to write clustering results to
  dump value = style filename
    style = standard or opendx
    filename = file to write viz data to
```

Examples:

```
diag_style cluster
diag_style cluster stats no delt 1.0 filename cluster3d.a.0.1.dat dump opendx cluster3d.a.0.1.dump
```

Description:

The cluster diagnostic computes a clustering analysis on all lattice sites in the system, identifying geometric groupings of identical spin values, e.g. a grain in a grain growth model. The total number of clusters is printed as stats output via the stats command.

Clustering uses a connectivity definition provided by the application (e.g. sites are adjacent and have same spin value) to identify the set of connected clusters.

The variants cluster, cluster2d, and cluster3d are used with applications based on lattice, lattice2d, and lattice3d, respectively.

The *filename* keyword allows an output file to be specified. Every time the cluster analysis is performed, the key properties of each cluster are appended to this file. The output format is:

Clustering Analysis for Lattice (diag_style cluster) nglobal = nprocs =

−− Time = ncluster = id ivalue dvalue size

cluster_id is an arbitrary integer assigned uniquely to each cluster. It will be different for different numbers of processors.

ivalue is an application−specific integer associated with each cluster. For lattice applications, it is the spin value of all sites in the cluster dvalue is an application−specific double associated with each cluster. size is the numbers of sites in the cluster.

The *dump* keyword causes the cluster ID for each site to be printed out in snapshot format which can be used for visualization purposes. The cluster IDs are arbitrary integers such that two sites have the same ID if and only if

they belong to the same cluster. The *standard* setting generates LAMMPS−style. For *cluster2d* and *cluster3d* styles only two values are printed for each site: site index and cluster ID. For the *cluster* style, three additional values are printed: the x, y, and z coordinate of the site (for 2d lattices, z=0). These files can be visualized with various tools in the LAMMPS package and the Pizza.py package.

The *opendx* keyword generates a set of files that can be read by the OpenDX script called aniso0.net to visualize the clusters in 3D. The filenames are composed of the input filename, followed by a sequential number, followed by '.dx'. Because the OpenDX format assumes a particular ordering of the sites, the *opendx* style can only be used with square and simple cubic lattices.

Restrictions:

This diagnostic can only be used for on−lattice applications.

Applications need to provide push_connected_neighbors() and connected_ghosts() functions which are called by this diagnostic. If they are not defined, SPPARKS will print an error message.

Related commands:

diag_style, stats

Default: none

diag_style diffusion command

Syntax:

```
diag_style diffusion keyword value keyword value ...
```

- diffusion = style name of this diagnostic
- see the diag_style command for additional keywords that can be appended to a diagnostic command

Examples:

```
diag_style diffusion
```

Description:

The diffusion diagnostic calculates outputs various statistics about the different events that have occurred in a cummulative sense since the simulation began. These values are printed as stats output via the stats command.

There are 4 kinds of events tallied, not all of which may occur depending on the parameters used in defining the app_style diffusion model.

- successful deposition event
- failed deposition event
- 1st neighbor hop
- 2nd neighbor hop

A successful deposition event is one that resulted in an atom added to the lattice. A failed deposition event is one that was attempted, but no suitable site could be found and thus no atom was added. A 1st neighbor hop is a diffusion hop from a lattice site to a nearest−neighbor vacancy. A 2nd neighbor hop is a Schwoebel hop from a lattice site to a 2nd nearest−neighbor vacancy. See the app_style diffusion command for more info on how Schwoebel hops occur.

Restrictions:

This diagnostic can only be used with the app_style diffusion application.

Related commands:

diag_style, stats

Default: none

diag_style energy command

Syntax:

```
diag_style energy keyword value keyword value ...
```

- energy = style name of this diagnostic
- see the diag_style command for additional keywords that can be appended to a diagnostic command

Examples:

```
diag_style energy
```

Description:

The energy diagnostic computes the total energy of all lattice sites in the system. The energy is printed as stats output via the stats command.

Restrictions:

This diagnostic can only be used for on−lattice applications.

Related commands:

diag_style, stats

Default: none

diag_style eprof3d command

Syntax:

```
diag_style eprof3d keyword value keyword value ...
```

- eprof3d = style name of this diagnostic
- zero or more keyword/value pairs may be appended
- see the diag_style command for additional keywords that can be appended to a diagnostic command and which must appear before these keywords
- keyword = *axis* or *filename* or *boundary*

```
  axis value = x or y or z
      x,y,z = which axis to measure energy profile with respect to
    filename value = name
      name = name of file to write results to
    boundary value = none
```

Examples:

```
diag_style eprof3d stats no delt 0.1 axis x filename eprof3d.dat
diag_style eprof3d filename eprof3d.dat boundary
```

Description:

The eprof3d diagnostic computes a one−dimensional average energy profile for all the lattice sites in the system.

The *axis* keyword specifices which axis to use as the profile coordinate.

The *filename* keyword allows a file to be specified which output is written to.

If the *boundary* keyword is used, the average energy is provided as a function of distance from the nearest sector boundary. In this case, the overall average energy and the average energy immediately to the left and right of the sector boundary is printed as stats output via the stats command. Also, in this case, the *axis* keyword has no effect.

If the *boundary* keyword is not used, then only the overall average energy is printed as stats output via the stats command.

Restrictions:

As described by the app_style command, on−lattice applications use one of 3 styles of lattice: general, 2d, or 3d. For this diagnostic only applications on 3d lattices are currently supported.

Related commands:

diag_style, stats

Default: none

diag_style propensity command

Syntax:

```
diag_style propensity keyword value keyword value ...
```

- propensity = style name of this diagnostic
- see the diag_style command for additional keywords that can be appended to a diagnostic command

Examples:

```
diag_style propensity
```

Description:

The propensity diagnostic computes the total propensity of all lattice sites in the system. The propensity is printed as stats output via the stats command.

The propensity can be thought of as the relative probablity of a site site to perform a KMC event. Note that if you are doing Metropolis MC and not kinetic MC, no propensity is defined.

Restrictions:

This diagnostic can only be used for on−lattice applications.

This diagnostic can only be used for KMC simulations where a solver is defined.

Related commands:

diag_style, stats

Default: none

diag_style command

Syntax:

```
diag_style style keyword value keyword value ...
```

- style = *cluster* or *diffusion* or *energy* or *eprof* or *propensity*
- zero or more keyword/value pairs may be appended

```
keyword = stats or delay or delt or logfreq
  stats values = yes or no
    yes/no = provide output to stats line
  delay values = tdelay
    tdelay = delay evaluating diagnostic until at least this time
  delt values = delta
    delta = time increment between evaluations of the diagnostic (seconds)
  logfreq values = N factor
    N = number of repetitions per interval
    factor = scale factor between interval
```

- see doc pages for individual diagnostic commands for additional keywords − diagnostic−specific keywords must come after any other standard keywords

Examples:

```
diag_style cluster stats no delt 1.0
diag_style eprof stats no delt 0.01 logfreq 7 10.0
diag_style energy
```

Description:

This command invokes a diagnostic calculation. Currently, diagnostics can only be defined for on−lattice applications. See the app_style command for an overview of such applications.

The diagnostics currently available are:

- cluster = grain size statistics for general lattices
- diffusion = statistics on diffusion events
- energy = energy of entire system for general lattices
- eprof = 1d energy profile
- propensity = propensity of entire system for general lattices

---

Diagnostics may provide one or more values that are appended to other statistical output and printed to the screen and log file via the stats command. This is stats output. In addition, the diagnostic may write more extensive output to its own files if requested by diagnostic−specific keywords.

The *stats* keyword controls whether or not the diagnostic appends values to the statistical output. If *stats* is set to *yes*, then none of the other keywords can be used, since the frequency of the stats output will determine when the diagnostic is called.

If *stats* is set to *no*, then the other keywords can be used, since presumably the diagnostic will create its own output files. The *delt* keyword specificies *Delta* = the interval of time between each diagnostic calculation. Similarly, the *logfreq* keyword will cause the diagnostic to run at varying intervals during the course of a

simulation. There will be *N* outputs per interval where the size of each interval scales up by *factor* each time. *Delta* is the time between outputs in the first (smallest) interval.

For example, this command

```
diag_style energy stats no delt 0.1 logfreq 7 10.0
```

will perform its computation at these times:

```
t = 0, 0.1, 0.2, ..., 0.7, 1, 2, ...., 7, 10, 20, ....
```

This command

```
diag_style energy stats no delt 0.1 logfreq 1 2.0
```

will perform its computation at these times:

```
t = 0, 0.1, 0.2, 0.4, 0.8, 1.6, ...
```

The delay keyword specifies the shortest time at which the diagnostic can be evaluated. This is useful if it is inconvenient to evaluate the diagnostic at time t=0.

Restrictions: none

Related commands:

stats

Default:

The stats setting is yes.

dimension command

Syntax:

```
dimension N
```

- N = 1 or 2 or 3

Examples:

```
dimension 2
```

Description:

Set the dimensionality of the simulation for spatial on−lattice or off−lattice models. By default SPPARKS runs 3d simulations. To run a 1d or 2d simulation, this command should be used prior to setting up a simulation box via the create_box or read_sites commands.

Restrictions:

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

Related commands: none

Default:

```
dimension 3
```

dump command

Syntax:

```
dump dump-ID delta filename field1 field2 ...
```

- dump−ID = user−assigned name for the dump
- delta = time increment between dumps (seconds)
- filename = name of file to dump snapshots to
- zero or more fields may be appended
- field = *id* or *site* or *x* or *y* or *z* or *energy* or *propensity* or *iN* or *dN*

Examples:

```
dump 1 0.25 tmp.dump
dump mydump 5.0 snap.ising id site energy i1
```

Description:

Dump snapshots of site values to a file at time intervals of *delta* during a simulation. As described below, the filename determines the kind of output (text or binary or gzipped, one big file or one per timestep, one big file or one per processor). The quantities printed are obtained from the application. Only on−lattice and off−lattice applications support dumps since they are spatial in nature. More that one dump command and file can be used during a simulation by giving each a unique dump−ID. Note that if written in appropriate format, a snapshot from a dump file can easily be converted into a data file suitable for input via the read_sites command to restart a simulation.

IMPORTANT NOTE: When running in parallel, the order of sites as printed to the dump file will be in chunks by processor, not ordered by ID. The order will be the same in every snapshot.

The dump_modify command can be used to alter the times at which snapshots are written out as well as defined a subset of sites to write out.

The text−based dump file is in the format of a LAMMPS dump file which can thus be read−in by the Pizza.py toolkit, converted to other formats, or used for visualization. An important modification to the LAMMPS−style header for each snapshot is the addition of real time to the line containing the snapshot number, i.e.

```
ITEM: TIMESTEP TIME
100   3.23945
```

The entry for "NUMBER OF ATOMS" is really number of sites, and will reflect any reduction in site count due to the dump_modify command, i.e.

```
ITEM: NUMBER OF ATOMS
314159
```

If fields are listed, then only those quantities will be printed for each site. If no fields are listed, then the default output values for each site are "id site x y z". These are the possible field values which may be specified.

The *id* is a unique integer ID for each site.

The *site*, *iN*, and *dN* fields specify a per−site value. *Site* is the same as *i1*. *iN* fields are integer values; *dN* fields are floating−point value. The application defines how many integer and floating=point values are stored for each site.

The *x*, *y*, *z* values are the coordinates of the site.

The *energy* value is what is computed by the energy() function in the application. Likewise for the *propensity* value which can be thought of as the relative probablity for that site to perform a KMC event. Note that if the application only performs rejection KMC or Metropolis MC, then no propensity is defined.

The specified filename determines how the dump file(s) is written. The default is to write one large text file, which is opened when the dump command is invoked and closed when an undump command is used or when SPPARKS exits.

Dump filenames can contain two wild−card characters. If a "*" character appears in the filename, then one file per snapshot is written and the "*" character is replaced with the timestep value. For example, tmp.dump.* becomes tmp.dump.0, tmp.dump.10000, tmp.dump.20000, etc.

If a "%" character appears in the filename, then one file is written for each processor and the "%" character is replaced with the processor ID from 0 to P−1. For example, tmp.dump.% becomes tmp.dump.0, tmp.dump.1, ... tmp.dump.P−1, etc. This creates smaller files and can be a fast mode of output on parallel machines that support parallel I/O for output.

Note that the "*" and "%" characters can be used together to produce a large number of small dump files!

If the filename ends with ".bin", the dump file (or files, if "*" or "%" is also used) is written in binary format. A binary dump file will be about the same size as a text version, but will typically write out much faster. Of course, when post−processing, you will need to convert it back to text format, using your own code to read the binary file. The format of the binary file can be understood by looking at the src/dump.cpp file.

If the filename ends with ".gz", the dump file (or files, if "*" or "%" is also used) is written in gzipped format. A gzipped dump file will be about 3x smaller than the text version, but will also take longer to write.

Restrictions:

This command can only be used as part of on−lattice or off−lattice applications. See the app_style command for further details.

To write gzipped dump files, you must compile SPPARKS with the −DSPPARKS_GZIP option − see the Making SPPARKS section of the documentation.

Related commands:

dump_one, dump_modify, undump, stats

Default: none

dump_modify command

Syntax:

```
dump_modify dump−ID keyword values ...
```

- dump−ID = ID of dump to modify
- one or more keyword/value pairs may be appended
- keyword = *delay* or *delta* or *flush* or *logfreq* or *thresh*

```
  delay value = tdelay
      tdelay = delay dump until at least this time (seconds)
    delta arg = dt
      dt = time increment between dumps (seconds)
    flush arg = yes or no
    logfreq values = N factor
      N = number of repetitions per interval
      factor = scale factor between interval
    thresh args = attribute operation value
      attribute = same fields (id,lattice,x,etc) used by dump command
      operation = "" or ">=" or "==" or "!="
      value = numeric value to compare to
      these 3 args can be replaced by the word "none" to turn off thresholding
```

Examples:

```
dump_modify 1 delay 30.0
dump_modify 1 logfreq 7 10.0 delay 100.0 flush yes
dump_modify mine thresh energy > 0.0 thresh id <= 1000
```

Description:

Dump snapshots of the state of the lattice to a file at intervals of delta during a simulation. The quantities printed are obtained from the application. Only lattice−based applications support dumps since what is output is one line per lattice site.

The *delay* keyword will suppress output until *tdelay* time has elapsed.

The *delta* keyword will suppress output until *tdelay* time has elapsed.

The *flush* option determines whether a flush operation in invoked after a dump snapshot is written to the dump file. A flush insures the output in that file is current (no buffering by the OS), even if SPPARKS halts before the simulation completes.

The *logfreq* keyword will produce statistical output at varying intervals during the course of a simulation. There will be *N* outputs per interval where the size of each interval is initially *delta* and then scales up by *factor* each time.

For example, this command

```
dump_modify 1 logfreq 7 10.0
```

will dump snapshots at these times:

```
t = 0, 0.1, 0.2, ..., 0.7, 1, 2, ...., 7, 10, 20, ....
```

This command

```
dump_modify mine logfreq 1 2.0
```

will dump snapshots at these times:

```
t = 0, 0.1, 0.2, 0.4, 0.8, 1.6, ...
```

If *N* is specified as 0, then this will turn off logarithmic output, and revert to regular output every *delta* seconds.

The *thresh* keyword allows sub−selection of lattice sites to output. Multiple thresholds can be specified. Specifying "none" turns off all threshold criteria. If thresholds are specified, only sites whose attributes meet all the threshold criteria are written to the dump file. The possible attributes that can be tested for are the same as the fields that can be specified in the dump command. Note that different attributes can be output by the dump command than are used as threshold criteria by the dump_modify command. E.g. you can output the coordinates and propensity of sites whose energy is above some threshold.

Restrictions:

This command can only be used as part of the lattice−based applications. See the app_style command for further details.

Related commands:

dump

Default:

The option defaults are delay = 0.0, delta = whatever was used in the dump command, flush = yes, logfreq = off, and thresh = none.

dump_one command

Syntax:

```
dump_one dump−ID
```

- dump−ID = ID of previously defined dump

Examples:

```
dump_one mine
dump_one 2
```

Description:

Dump the current state of the lattice to the dump file defined by the dump command with this *dump−ID*. This can be useful before or after a run, if the dump command itself did not produce a snapshot at the desired time or state.

The information dumped is determined by the dump command which must have been previously specified to use the dump_one command.

Restrictions:

This command cannot be used to trigger the very first snapshot written to the file specified with the dump command.

Related commands:

dump

Default: none

echo command

Syntax:

```
echo style
```

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

Examples:

```
echo both
echo log
```

Description:

This command determines whether SPPARKS 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.

Restrictions: none

Related commands: none

Default:

```
echo log
```

ecoord command

Syntax:

```
ecoord N eng
```

- N = coordination number (see asterisk form below)
- eng = energy of site with this coordination number (energy units)

Examples:

```
ecoord 8 5.6
ecoord 0 1.0e20
ecoord * 1.0
ecoord 8*12 10.0
```

Description:

This command sets the energy of an occupied site in a lattice as a function of coordination number, where coordination = the number of occupied neighbor sites. See the app_style diffusion nonlinear command for how the energy change of the system due to a diffusive hop is used to calculate a probability for the hop to occur.

Typically, Nmax+1 values should be specified by using this command one or more times, with N varying from 0 to Nmax, when Nmax is the number of neighbor sites for each lattice site.

The N index can be specified in one of two ways. An explicit numeric value can be used, as in the 1st example above. Or a wild−card asterisk can be used to set the energy value for multiple coordination numbers. This takes the form "*" or "*n" or "n*" or "m*n". If Nmax = the number of neighbor sites, then an asterisk with no numeric values means all coordination numbers from 0 to Nmax. A leading asterisk means all coordination numbers from 0 to n (inclusive). A trailing asterisk means all coordination numbers from n to Nmax (inclusive). A middle asterisk means all coordination numbers from m to n (inclusive).

Note that if the third example is specfied first, followed by the first example, then the effect would be to set the energy value for all coordination numbers to 1.0, then overwrite the energy value for coordination number 8 to 5.6.

The *eng* value should be in the energy units defined by the application's Hamiltonian and should be consistent with the units used in any temperature command.

Restrictions:

This command can only be used as part of the app_style diffusion nonlinear application.

Related commands:

deposition, barrier

Default:

Energy values for all coordination numbers are set to 0.

if command

Syntax:

```
if value1 operator value2 then command1 else command2
```

- value1 = 1st value
- operator = "" or ">=" or "==" or "!="
- value2 = 2nd value
- then = required word
- command1 = command to execute if condition is met
- else = optional word
- command2 = command to execute if condition is not met (optional argument)

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 then "timestep 0.005"
if ${eng} > ${eng_previous} then "jump file1" else "jump file2"
```

Description:

This command provides an in−then−else test capability within an input script. Two values are numerically compared to each other and the result is TRUE or FALSE. Note that as in the examples above, either of the values can be variables, as defined by the variable command, so that when they are evaluated when substituted for in the if command, a user−defined computation will be performed which can depend on the current state of the simulation.

If the result of the if test is TRUE, then command1 is executed. This can be any valid SPPARKS input script command. If the command is more than 1 word, it should be enclosed in double quotes, so that it will be treated as a single argument, as in the examples above.

The if command can contain an optional "else" clause. If it does and the result of the if test is FALSE, then command2 is executed.

Note that if either command1 or command2 is a bogus SPPARKS command, such as "exit" in the first example, then executing the command will cause SPPARKS to halt.

Restrictions: none

Related commands:

variable

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 SPPARKS 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 SPPARKS 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

inclusion command

Syntax:

```
inclusion x y z r
```

- x,y,z = position of center of protein inclusion
- r = radius of the protein

Examples:

```
inclusion 10 12 0.0 2.0
inclusion 10 12 5.4 5.0
```

Description:

This command defines protein sites on a lattice and can only be used by app_style membrane applications.

Think of the protein as a sphere (or circle) centered at *x*,*y*,*z* and with a radius of *r*. All lattice sites within the sphere (or circle) will be flagged as protein (as opposed to lipid or solvent). For lattices with a 2d geometry, the z value should be speficied as 0.0.

Restrictions: none

This command can only be used as part of the app_style pore applications.

Related commands:

app_style membrane

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
```

Description:

This command closes the current input script file, opens the file with the specified name, and begins reading SPPARKS commands from that file. 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.

Optionally, if a 2nd argument is used, 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
run 5.0
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, SPPARKS 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
```

Restrictions:

If you jump to a file and it does not contain the specified label, SPPARKS 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

lattice command

Syntax:

```
lattice style args
```

- style = *line/2n* or *sq/4n* or *sq/8n* or *tri* or *sc/6n* or *sc/26n* or *bcc* or *fcc* or *diamond* or *fcc/octa/tetra* or *random/1d* or *random/2d or random/3d*

```
  arg for all styles except random = scale
    scale = lattice constant (distance units)

random/1d args = Nrandom cutoff
  random/2d args = Nrandom cutoff
  random/3d args = Nrandom cutoff
    Nrandom = # of random sites
    cutoff = distance within which sites are connected (distance units)
```

Examples:

```
lattice sq/4n 1.0
lattice fcc 3.52
lattice random/3d 10000 2.0
```

Description:

Define a lattice for use by other commands. In SPPARKS, a lattice is simply a set of points in space, determined by a unit cell with basis atoms, that is replicated infinitely in all dimensions. The arguments of the lattice command can be used to define a wide variety of crystallographic lattices.

A lattice is used by SPPARKS in two ways. First, the create_sites command creates "sites" on the lattice points inside the simulation box. Sites are used by an on−lattice or off−lattice application, specified by the app_style command, which define events that change the values associated with sites (e.g. a spin flip) or the coordinates of the site itself (for off−lattice applications).

Second, the lattice spacing in the x,y,z dimensions is used by other commands such as the region command to define distance units and define geometric extents, for example in specifying the size of the simulation box via the create_box command.

The lattice style must be consistent with the dimension of the simulation − see the dimension command and descriptions of each style below.

A lattice consists of a unit cell, a set of basis sites within that cell. The vectors a1,a2,a3 are the edge vectors of the unit cell. This is the nomenclature for "primitive" vectors in solid−state crystallography, but in SPPARKS the unit cell they determine does not have to be a "primitive cell" of minimum volume.

For on−lattice applications (see the app_style command), the lattice definition also infers a connectivity between lattice sites, which is used to generate the list of neighbors of each site. This information is ignored for off−lattice applications. This means that for a 2d off−lattice application, it makes no difference whether a *sq/4n* or *sq/8n* lattice is used; they both simply generate a square lattice of points.

In the style descriptions that follow, a = the lattice constant defined by the lattice command. Sites within a unit cell are defined as (x,y,z) where 0.0 <= x,y,z < 1.0.

A lattice of style *line/2n* is a 1d lattice with a1 = a 0 0 and one basis site per unit cell at (0,0,0). Each lattice point has 2 neighbors.

Lattices of style *sq/4n* and *sq/8n* are 2d lattices with a1 = a 0 0 and a2 = 0 a 0, and one basis site per unit cell at (0,0,0). The *sq/4n* style has 4 neighbors per site (east/west/north/south); the *sq/8n* style has 8 neighbors per site (same 4 as *sq/4n* plus 4 corner points).

A lattice of style *tri* is a 2d lattice with a1 = a 0 0 and a2 = 0 sqrt(3)*a 0, and two basis sites per unit cell at (0,0,0) and (0.5,0.5,0). Each lattice points has 6 neighbors.

Lattices of style *sc/6n* and *sc/26n* are 3d lattices with a1 = a 0 0 and a2 = 0 a 0 and a3 = 0 0 a, and one basis site per unit cell at (0,0,0). The *sc/6n* style has 6 neighbors per site (east/west/north/south/up/down); the *sc/26n* style has 26 neighbors per site (surrounding cube including edge and corner points).

Lattices of style *bcc* and *fcc* and *diamond* are 3d lattice with a1 = a 0 0 and a2 = 0 a 0 and a3 = 0 0 a. There are two basis sites per unit cell for *bcc*, 4 basis sites for *fcc*, and 8 sites for *diamond*. The location of the basis sites are defined in any solid−state physics or crystallography text. The *bcc* style has 8 neighbors per site, the *fcc* has 12, and the *diamond* has 4.

A lattice of style *fcc/octa/tetra* is a 3d lattice with a1 = a 0 0 and a2 = 0 a 0 and a3 = 0 0 a. There are 16 basis sites per unit cell, which consist of 4 fcc sites plus 4 octahedral and 8 tetrahedral interstitial sites. Again, these are defined in solid−state physics texts. There are 26 neighbors per fcc and octahedral site, and 14 neihbors per tetrahedral site.

The *random* lattice styles are 1d, 2d, and 3d lattices with a1 = 1 0 0 and a2 = 0 1 0 and a3 = 0 0 1. Note that no *scale* parameter is defined and the unit cell is a unit cube, not a cube with side length *a*. Thus a region command using one of these lattices will define its geometric region directly, not as multiples of the *scale* parameter. When the create_sites command is used, it will generate a collection of Nrandom points within the corresponding 1d, 2d, or 3d region or simulation box. The number of neighbors per site is defined by the specified *cutoff* parameter. Two sites I,J will be neighbors of each other if they are closer than the *cutoff* distance apart.

Restrictions: none

Related commands:

dimension, create_sites, region

Default: none

log command

Syntax:

```
log file
```

- file = name of new logfile

Examples:

```
log log.equil
```

Description:

This command closes the current SPPARKS 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.spk" is the default log file for a SPPARKS 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 SPPARKS log file is named log.spk

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 SPPARKS 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. *Equal*− or *world*−style variables cannot be incremented by a next command. All the variables specified are incremented by one value from their respective lists.

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.

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 SPPARKS 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.spparks.variable" and "tmp.spparks.variable.lock" which you will see in your directory during such a SPPARKS 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
```

Restrictions: none

Related commands:

jump, include, shell, variable,

Default: none

pair_coeff command

Syntax:

```
pair_coeff I J args ...
```

- I,J = atom types (see asterisk form below)
- args = coefficients for one or more pairs of atom types

Examples:

Examples:

```
pair_coeff 1 2 1.0 1.0 2.5
pair_coeff 2 * 1.0 1.0
```

Description:

Specify the pairwise force field coefficients for one or more pairs of atom types. The number and meaning of the coefficients depends on the pair style.

I and J can be specified in one of two ways. Explicit numeric values can be used for each, as in the 1st example above. I <= J is required. SPPARKS sets the coefficients for the symmetric J,I interaction to the same values.

A wild−card asterisk can be used in place of or in conjunction with the I,J arguments to set the coefficients for multiple pairs of atom types. This takes the form "*" or "*n" or "n*" or "m*n". If N = the number of atom types, then an asterisk with no numeric values means all types from 1 to N. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to N (inclusive). A middle asterisk means all types from m to n (inclusive). Note that only type pairs with I <= J are considered; if asterisks imply type pairs where J < I, they are ignored.

Note that a pair_coeff command can override a previous setting for the same I,J pair. For example, these commands set the coeffs for all I,J pairs, then overwrite the coeffs for just the I,J = 2,3 pair:

```
pair_coeff * * 1.0 1.0 2.5
pair_coeff 2 3 2.0 1.0 1.12
```

For many potentials, if coefficients for type pairs with I != J are not set explicitly by a pair_coeff command, the values are inferred from the I,I and J,J settings by mixing rules. Details on the mixing as it pertains to individual potentials are described on the doc page for the potential.

Here is the list of pair styles defined in SPPARKS. More will be added as new applications are developed. Click on the style to display the formula it computes, arguments specified in the pair_style command, and coefficients specified by the associated pair_coeff command:

- pair_style lj/cut − cutoff Lennard−Jones potential

Restrictions: none

Related commands:

pair_style

Default: none

pair_style lj command

Syntax:

```
pair_style lj Ntypes cutoff
```

- lj = style name of this pair style
- Ntypes = # of particle types
- cutoff = global cutoff for pairwise interactions (distance units)

Examples:

```
pair_style lj 1 2.5
pair_style lj 3 3.0
```

Description:

The *lj/cut* style computes the standard 12/6 Lennard−Jones potential, given by

$$E = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \qquad r < r_c$$

Rc is the cutoff.

The following coefficients must be defined for each pair of particle types via the pair_coeff command, or by mixing as described below:

- epsilon (energy units)
- sigma (distance units)
- cutoff (distance units)

Note that sigma is defined in the LJ formula as the zero−crossing distance for the potential, not as the energy minimum at 2^(1/6) sigma.

The last coefficients is optional. If not specified, the global LJ cutoff specified in the pair_style command is used.

Mixing info:

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for all of the lj/cut pair styles can be mixed. The style of mixing is *geometric*, which means that

```
epsilon_ij = sqrt(epsilon_i * epsilon_j)
sigma_ij = sqrt(sigma_i * sigma_j)
```

Restrictions: none

Related commands: none

Default: none

pair_style command

Syntax:

```
pair_style style args ...
```

- style = one of the styles from the list below
- args = arguments used by a particular style

Examples:

```
pair_style lj 1 2.5
```

Description:

Set the formula(s) SPPARKS uses to compute pairwise energy of interaction between sites or particles in an off−lattice application.

The coefficients associated with a pair style are typically set for each pair of particle types, and are specified by the pair_coeff command.

Here is the list of pair styles defined in SPPARKS. More will be added as new applications are developed. Click on the style to display the formula it computes, arguments specified in the pair_style command, and coefficients specified by the associated pair_coeff command:

- pair_style lj/cut − cutoff Lennard−Jones potential

Restrictions: none

Related commands:

pair_style

Default: none

pin command

Syntax:

```
pin fraction multiflag nthresh
```

- fraction = fraction of sites (0 to 1) to convert to pinned sites
- multiflag = 0 for single sites, 1 for sites+neighbors
- nthresh = # of neighbor sites which must have different spins

Examples:

```
pin 0.1 0 2
```

Description:

This command converts sites on a lattice to pinned sites by setting their spin value to Q+1, where Q is defined by a Potts model. This command can only be used by the app_style potts/pin application. The size of the inclusions and their location (anywhere or preferentially near grain boundaries) can be controlled by the *multiflag* and *nthresh* parameters.

The way pinned sites are selected is as follows. A pinned site is chosen randomly. If the site is already a pinned site, then another site is selected. If *multiflag* is set to 1, then if any of the site's neighbors are already a pinned site, then another site is selected. If *nthresh* is a non−zero value, then the # of neighbor sites with spin values different than the chosen site are counted. If the count is less than *nthresh*, then another site is selected.

Once the site is selected, just that site is converted to a pinned site if *multiflag* is 0. If *multiflag* is 1, then the site plus all its neibhbors are converted to pinned sites.

This process continues until the desired fraction of changed sites is achieved. The entire process is done in a way that should be independent of the number of processors used to run a particular simulation.

Note that if you pick a large volume fraction and/or a high value for *nthresh* it is possible that SPPARKS will never find enough valid sites to convert to pinned sites. It will then loop endlessly.

Restrictions: none

This command can only be used as part of the app_style potts/pin applications.

Related commands:

app_style potts/pin

Default: none

print command

Syntax:

```
print string
```

- string = text string to print. may contain variables

Examples:

```
print "Done with equilibration"
print "The system temperature is now $t"
```

Description:

Print a text string to the screen and logfile. The text string must be a single argument, so it should be enclosed in double quotes if it is more than one word. 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) then 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, or references to other variables.

Restrictions: none

Related commands:

variable

Default: none

processors command

Syntax:

```
processors Px Py Pz
```

- Px,Py,Pz = # of processors in each dimension of a 3d grid

Examples:

```
processors 2 4 4
```

Description:

Specify how processors are mapped as a 3d logical grid to the global simulation box for spatial on−lattice or off−lattice models.

When this command has not been specified, SPPARKS will choose Px, Py, Pz based on the dimensions of the global simulation box so as to minimize the surface/volume ratio of each processor's sub−domain.

Since SPPARKS does not load−balance by changing the grid of 3d processors on−the−fly, this command should be used to override the SPPARKS default if it is known to be sub−optimal for a particular problem.

The product of Px, Py, Pz must equal P, the total # of processors SPPARKS is running on. 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.

If P is large and prime, a grid such as 1 x P x 1 will be required, which may incur extra communication costs.

Restrictions:

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

Related commands: none

Default:

SPPARKS chooses Px, Py, Pz

read_sites command

Syntax:

```
read_sites file
```

- file = name of data file to read in

Examples:

```
read_sites data.potts
read_sites ../run7/data.potts.gz
```

Description:

Read in a data file containing information SPPARKS needs to setup an on−lattice or off−lattice application. The file can be ASCII text or a gzipped text file (detected by a .gz suffix). This is one of 2 ways to specify event sites ; see the create_sites command for another method.

A data file has a header and a body, as described below. The body of the file contains up to 3 sections: Sites, Neighbors, Values. Sites defines the coordinates of event sites. Neighbors define the neighbors of each site (only for on−lattice applications). Values assign per−site values to each site, which can also be done via the set command.

If a simulation box has not already been created, then the data file defines the box size (in the header), and it must define Sites and Neighbors (for on−lattice applications). The Values section is optional.

If a simulation box has already been defined (by the "create_box" command or a previous read_sites command), but no sites have previosly been defined, then the box size in the header must be the same, and the data file must define Sites and Neighbors (for on−lattice applications). The Values section is optional.

If a simulation box has already been defined, and sites have previosly been defined (by the "create_sites" command or a previous read_sites command), then the box size in the header must be the same, no Sites or Neighbors can be specfied, but the Values section may be used. This is a means of restarting a simulation using per−site info written out by the dump command and reformatted so it can be input by this command.

The first line of the header of the data file is always skipped; it typically contains a description of the file. Then lines are read one at a time. Lines can have a trailing comment starting with '#' that is ignored. If the line is blank (only whitespace after comment is deleted), it is skipped. If the line contains a header keyword, the corresponding value(s) is read from the line. If it doesn't contain a header keyword, the line begins the body of the file.

The body of the file contains zero or more sections. The first line of a section has only a keyword. The next line is skipped. The remaining lines of the section contain values. The number of lines depends on the section keyword as described below. Zero or more blank lines can be used between sections. Sections can appear in any order, with a few exceptions as noted below.

The formatting of individual lines in the data file (indentation, spacing between words and numbers) is not important except that header and section keywords (e.g. dimension, xlo xhi, Sites, Values) must be capitalized as shown and can't have extra white space between their words − e.g. two spaces or a tab between "xlo and "xhi" is not valid.

These are the recognized header keywords. Header lines can come in any order. The value(s) are read from the beginning of the line. Thus the keyword *sites* should be in a line like "1000 sites"; the keyword *ylo yhi* should be in a line like "−10.0 10.0 ylo yhi". All these settings have a default value of 0, except the lo/hi box size defaults are −0.5 and 0.5. A line need only appear if the value is different than the default.

- *dimension* = dimension of system = 1,2,3
- *sites* = number of sites
- *max neighbors* = max # of neighbors of any site
- *xlo xhi* = simulation box boundaries in x dimension
- *ylo yhi* = simulation box boundaries in y dimension
- *zlo zhi* = simulation box boundaries in z dimension

The *max neighbors* setting is only needed if the file contains a Neighbors section, which is only used for on−lattice applications.

The simulation box size is determined by the lo/hi settings. For 2d simulations, the *zlo zhi* values should be set to bound the z coords for atoms that appear in the file; the default of −0.5 0.5 is valid if all z coords are 0.0. The same rules hold for *ylo and yhi* for 1d simulations.

These are the possible section keywords for the body of the file: *Sites, Neighbors, Values*.

Each section is listed below. The format of each section is described including the number of lines it must contain and rules (if any) for where it can appear in the data file.

Any individual line in the various sections can have a trailing comment starting with "#" for annotation purposes. E.g. in the Sites section:

10 10.0 5.0 6.0 # impuity site

Sites section:

- one line per site
- line syntax: ID x y z

```
ID = global site ID (1−N)
x y z = coordinates of site
```

- example:

```
101 7.0 0.0 3.0
```

There must be N lines in this section where N = number of sites. The lines can appear in any order.

Neighbors section:

- one line per site
- line syntax: ID n1 n2 n3 ...

```
ID = global site ID (1−N)
n1 n2 n3 ... = IDs of neighbor sites
```

- example:

```
101 7 32 15 1004 ...
```

There must be N lines in this section where N = number of sites. The lines can appear in any order.

The number of neighbors can vary from site to site, but there can be no more than *max neighbors* for any one site. The neighbors of an individual site can be listed in any order.

Values section:

- one line per site
- line syntax: ID i1 i2 ... iN d1 d2 ... dN

```
ID = global site ID (1−N)
i1,i2,...iN = integer values for the site
d1,d2,...dN = floating point values for the site
```

- example:

```
101 1 3 4.0
```

There must be N lines in this section where N = number of sites. The lines can appear in any order.

The number of values per site that should be listed depends on the application which defines the number of integer and floating−point values per site. These are listed in order, with the integer values first, followed by the floating−point values.

Restrictions:

To write gzipped dump files, you must compile SPPARKS with the −DSPPARKS_GZIP option − see the Making SPPARKS section of the documentation.

Related commands:

create_box, create_sites, set

Default: none

region command

Syntax:

```
region ID style args keyword value ...
```

- ID = user−assigned name for the region
- style = *block* or *cylinder* or *sphere* or *union* or *intersect*

```
  block args = xlo xhi ylo yhi zlo zhi
      xlo,xhi,ylo,yhi,zlo,zhi = bounds of block in all dimensions (distance units)
    cylinder args = dim c1 c2 radius lo hi
      dim = x or y or z = axis of cylinder
      c1,c2 = coords of cylinder axis in other 2 dimensions (distance units)
      radius = cylinder radius (distance units)
      lo,hi = bounds of cylinder in dim (distance units)
    sphere args = x y z radius
      x,y,z = center of sphere (distance units)
      radius = radius of sphere (distance units)
    union args = N reg−ID1 reg−ID2 ...
      N = # of regions to follow, must be 2 or greater
      reg−ID1,reg−ID2, ... = IDs of regions to join together
    intersect args = N reg−ID1 reg−ID2 ...
      N = # of regions to follow, must be 2 or greater
      reg−ID1,reg−ID2, ... = IDs of regions to intersect
```

- zero or more keyword/value pairs may be appended
- keyword = *side side* value = *in* or *out in* = the region is inside the specified geometry *out* = the region is outside the specified geometry

Examples:

```
region 1 block −3.0 5.0 INF 10.0 INF INF
region 2 sphere 0.0 0.0 0.0 5 side out
region void cylinder y 2 3 5 −5.0 EDGE
region outside union 4 side1 side2 side3 side4
```

Description:

This command defines a geometric region of space. Various other commands use regions. For example, the region can be filled with sites via the create_sites command.

The distance units used to define the region are setup by the lattice command which must be used before any regions are defined. The lattice command defines a lattice spacing and regions are defined in terms of this length scale. For example, if the lattice spacing is 3.0 and the region sphere radius is 2.5, then the size of the sphere is 2.5*3.0 = 7.5.

The lo/hi values for the *block* or *cylinder* styles can be specified as EDGE or INF. EDGE means they extend all the way to the global simulation box boundary. Note that this is the current box boundary; if the box changes size during a simulation, the region does not. INF means a large negative or positive number (1.0e20), so it should encompass the simulation box even if it changes size. If a region is defined before the simulation box has been created (via create_box or read_sites commands), then an EDGE or INF parameter cannot be used.

For style *cylinder*, the c1,c2 params are coordinates in the 2 other dimensions besides the cylinder axis dimension. For dim = x, c1/c2 = y/z; for dim = y, c1/c2 = x/z; for dim = z, c1/c2 = x/y. Thus the third example above

specifies a cylinder with its axis in the y−direction located at x = 2.0 and z = 3.0, with a radius of 5.0, and extending in the y−direction from −5.0 to the upper box boundary.

The *union* style creates a region consisting of the volume of all the listed regions combined. The *intersect* style creates a region consisting of the volume that is common to all the listed regions.

The *side* keyword determines whether the region is considered to be inside or outside of the specified geometry. Using this keyword in conjunction with *union* and *intersect* regions, complex geometries can be built up. For example, if the interior of two spheres were each defined as regions, and a *union* style with *side* = out was constructed listing the region−IDs of the 2 spheres, the resulting region would be all the volume in the simulation box that was outside both of the spheres.

Restrictions: none

Related commands:

lattice, create_sites

Default:

The option defaults are side = in.

reset_time command

Syntax:

```
reset_time time
```

- time = new time

Examples:

```
reset_time 0.0
reset_time 100.0
```

Description:

Set the current time to the specified value. This can be useful if a preliminary run was performed and you wish to reset the time before performing a subsequent run.

Restrictions: none

Related commands: none

Default: none

run command

Syntax:

```
run delta keyword values ...
```

- delta = run simulation for this amount of time (seconds)
- zero or more keyword/value pairs may be appended
- keyword = *upto* or *pre* or *post*

```
  upto value = none
    pre value = no or yes
    post value = no or yes
```

Examples:

```
run 100.0
run 10000.0 upto
run 1000 pre no post yes
```

Description:

This command runs a Monte Carlo application for the specified number of seconds of simulation time. If multiple run commands are used, the simulation is continued, possibly with new settings which were specified between the successive run commands.

The application defines Monte Carlo events and probabilities which determine the amount of physical time associated with each event.

A value of delta = 0.0 is acceptable; only the status of the system is computed and printed without making any Monte Carlo moves.

The *upto* keyword means to perform a run starting at the current time up to the specified time. E.g. if the current time is 10.0 and "run 100.0 upto" is used, then an additional 90.0 seconds will be run. This can be useful for very long runs on a machine that allocates chunks of time and terminate your job when time is exceeded. If you need to restart your script multiple times (reading in the last restart file), you can keep restarting your script with the same run command until the simulation finally completes.

The *pre* and *post* keywords can be used to streamline the setup, clean−up, and associated output to the screen that happens before and after a run. This can be useful if you wish to do many short runs in succession (e.g. SPPARKS is being called as a library which is doing other computations between successive short SPPARKS runs).

By default (pre and post = *yes*), SPPARKS initializes data structures and computes propensities before every run. After every run it gathers and prints timings statistics. If a run is just a continuation of a previous run, the data structure initialization is not necessary. So if *pre* is specified as *no* then the initialization is skipped. Propensities are still re−computed since commands between runs or a driver program may have changed the system, e.g. by altering lattice values. Note that if *pre* is set to *no* for the very 1st run SPPAKRS performs, then it is overridden, since the initialization must be done.

If *post* is specified as *no*, the full timing summary is skipped; only a one−line summary timing is printed.

Restrictions: none

Related commands: none

Default:

The option defaults are pre = yes and post = yes.

sector command

Syntax:

```
sector flag keyword value ...
```

- flag = *yes* or *no* or *N* where N = 2,4,8
- zero or more keyword/value pairs may be appended
- keyword = *tstop* or *nstop*

```
  tstop value = dt
      dt = elapsed time for events to perform within sector (seconds)
    nstop value = N
      N = average number of events per site to perform within sector
```

Examples:

```
sector no
sector yes
sector 4
sector yes nstop 0.5
sector yes tstop 5.0
```

Description:

This command partitions the portion of the simulation domain owned by each processor into sectors or sub−domains. It can only be used for on−lattice applications. Typically, it is used in a parallel simulation, to enable parallelism, but it can also be used on a single processor.

If sectoring is enabled via the *yes* setting, then for 1d lattices, each processor's sub−domain is partioned into 2 halves, for 2d lattices, each processor's sub−domain is partitioned into 4 quadrants, and for 3d lattices it is partitioned into 8 octants. If the *N* setting is used instead, then the number of sectors can be specified directly. This may be useful in some models to reduce communication. A 3d lattice can use 2 (x only) or 4 sectors (x and y), instead of the default 8 (x and y and z). A 2d lattice can use 2 sectors (x only), instead of the default 4 (x and y). Note that if no sectors are used in a dimension, then there must be only one processor assigned to that dimension of the simulation box (see the app_style procs command). For example, if "sector 2" is used for a 2d lattice, then the processor layout must be Px1, where P is the total number of processors.

If sectors are turned on, then a kinetic Monte Carlo (KMC) or rejection KMC (rKMC) algorithm is performed in the following manner. Events or sites are selected within the first sector on each processor, via a solver or sweeping method. Communication is then done between processors to update sector boundaries. Then all proecessors move to the next sector, and the process is repeated. Thus a single sweep over the entire lattice is performed in 2 (or 4 or 8) stages for 1d (of 2d or 3d) lattices, as sectors are processed one at a time, followed by the appropriate communication. This procedure insure events occurring on one processor do not conflict with events performed by other processors.

The optional keywords determine how much time is spent on each sector (i.e. how many events are performed) before moving to the next sector. See the discussion below for what they mean when sectoring is set to *no*.

Note that using sectors turns an exact KMC or rKMC algorithm into an approximate one, in the spirit of Amar. This is because events are occuring within a sector while the state of the system on the boundary of the sector is held frozen. If the time−per−sector is too large, this will require less communication but will induce incorrect

dynamics at the sector boundaries. Conversely, if the time−per−sector is too small, the simulation will perform few events per sector and spend too much time communicating.

If the *tstop* keyword is set to a value > 0.0, it sets the time per sector to the specified value. For a KMC algorithm, events are performed until this time threshhold is reached. For a rKMC algorithm, a time per attempted event is defined, and events are attempted until this time threshhold is reached.

If the *nstop* keyword is set to a value > 0.0, it sets the average number of events (or attempts) per site. For example, an *nstop* value of 2.0 means attempt 2 events per site for an rKMC algorithm. For a KMC algorithm, this is converted into a time by computing the maximum propensity of all sites within any sector in the simulation domain. In the KMC case, this means that if the total propensity of the system decreases as the simulation proceeds (e.g. grain growth occurs), then the effective time per sweep will increase in an adaptive way. Said another way, the number of events per sweep will remain roughly constant, as the time per event increases. In the rKMC case, the time per attempt is constant due to the use of a null−bin, so there is no adaptivity.

If neither the *tstop* or *nstop* keywords are specified, a default value of nstop = 1.0 is used, meaning one event per site is performed or attempted in the KMC or rKMC algorithm in each sector. This should give good behavior in many applications, meaning high accuracy is achieved with good parallel performance due to a modest amount of communication being performed.

Note that it makes no sense to specify both *tstop* and *nstop* since they define the time−per−sector in different ways. When *tstop* is specified, it sets *nstop* to 0.0. Likewise when *nstop* is specified, it sets *tstop* to 0.0. Thus if both are used, the last setting takes precedence.

If sectors are turned off via the *no* setting, then the *nstop* or *tstop* settings still have an effect for rKMC simulations where the sweep style is set to *color*. They determine how many times the sites associated with each color are looped over before moving to the next color. Normally, this should just be 1, which is the *nstop* default, but this can be changed if desired.

Restrictions:

This command can only be used as part of on−lattice applications as specified by the app_style command.

Related commands:

app_style, solve_style, sweep

Default:

The default for sectoring is *no* and the option defaults are nstop = 1.0 and tstop = 0.0.

(Amar) Shin and Amar, Phys Rev B, 71, 125432−1−125432−13 (2005).

seed command

Syntax:

```
seed Nvalue
```

- Nvalue = seed for a random number generator (positive integer)

Examples:

```
seed 5838959
```

Description:

This command sets the random number seed for a master random number generator which is used by SPPARKS to initialize auxiliary random number generators which in turn are used for all operations in the code requiring random numbers. Thus this command is needed to perform any simulation with SPPARKS.

Restrictions: none

Related commands: none

Default: none

set command

Syntax:

```
set label style args keyword values ...
```

- label = *site* or *iN* or *dN* or *x* or *y* or *z* or *xyz*
- style = *value* or *range* or *displace*

```
value arg = nvalue
   nvalue = value to set sites to
range args = lo hi
  lo,hi = range of values to set sites to
displace arg = delta
   delta = max distance to displace the site
```

- zero or more keyword/value pairs may be appended
- keyword = *fraction* or *region* or *loop* or *if*

```
fraction value = frac
    frac = fractional value > 0 and <= 1.0
  region args = region-ID
    region-ID = ID of region that sites must be part of
  loop arg = all or local
    all = loop over all sites
    local = loop over only sites I own
  if args = label2 op nvalue2
    label2 = id or iN or dN or x or y or z
    op = "" or "<=" or "=" or "!="
    nvalue2 = value to compare site value to
```

Examples:

```
set i1 value 2 fraction 0.5
set d1 range 1.0 2.0 loop local
set xyz displace 0.2
set i1 range 1 50 if x <20 if i2 = 3
```

Description:

Reset a per−site value for one or more sites. Each on−lattice or off−lattice application defines what per−site values are stored with each site in its model. When sites are created by the create_sites or read_sites commands, their per−site values may be set to zero or to values specified by those commands. This command enables the values to be changed, either before the first run, or between runs.

The *label* determines which per−site quantity is set. *iN* and *dN* mean the Nth integer or floating−point quantity, with 1 <= N <= Nmax. Nmax is defined by the application. If *label* is specified as *site* it is the same as *i1*. For off−lattice applications, the *x* or *y* or *z* or *xyz* coordinates of each site can be adjusted.

For label *iN* or *dN* or *site*, the styles *value* or *range* can be used.

For style *value*, the per−site quantity is set to the specified *nvalue*, which should be either an integer or floating−point numeric value, depending on what kind of per−site quantity is being set.

For style *range*, the per−site quantity is set to a random value between *lo* and *hi* (inclusive). Both *lo* and *hi* should be either integer or floating−point numeric values, depending on what kind of per−site quantity is being set.

NOTE: The *displace* style is not yet implemented but will be soon. The following text explains how it will work for off−lattice applications.

For label *x* or *y* or *z* or *xyz*, the style *displace* must be used. For *x* or *y* or z, the corresponding coordinate of each site is displaced by a random distance between −*delta* and *delta*. For *xyz* the site is displaced to a new random point within a sphere of radius *delta* surrounding the site (or a circle for 2d models, or a line segement for 1d models).

The optional keywords enables selection of sites whose *label* quantity will be reset to a new value. Note that these optional keywords can be used in various combinations, and the *if* keyword can be used multiple times, to select desired sites.

The keyword *fraction* means that only a fraction of the sites will be reset, where 0 < *frac* <= 1.0. For each site a random number R is generated and the reset only occurs if R < *frac*.

The keyword *region* means that only sites in the specified region will be reset. Note that a defined region can be a union or intersection of several regions and can be either inside or outside a geometric boundary; see the region command for details.

The keyword *loop* determines how sites in the simulation box are looped over when their per−site quantity is reset. In general, each processor will own some subset Nlocal of the total number of sites Nglobal in the simulation box. The entire set of sites are assumed to have IDs from 1 to Nglobal. For *loop all*, each processor performs a loop from 1 to Nglobal and generates the new value for that site. If it owns the site, then it resets its value. This means that the changes to per−site values will be the same, independent of which processor owns which site. For *loop local*, each processor loops over only its sites from 1 to Nlocal. This may be faster, but if random numbers are used to determine new per−site values, it will give different answers depending on the the number of processors used.

The keyword *if* sets a condition that must be met in order for the per−site quantity to be reset. The per−site quantity specified by *label2* is compared to the numeric *nvalue2* and if the condition is not met, then the site is skipped.

Restrictions: none

Related commands:

create_sites, read_sites

Default:

The default values for the optional keywords is fraction 1.0 and loop all. No region is defined by default nor are any if−tests.

shell command

Syntax:

```
shell style args
```

- style = *cd* or *mkdir* or *mv* or *rm* or *rmdir*

```
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
```

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
```

Description:

Execute a shell command. Only a few simple file−based shell commands are supported, in Unix−style syntax. With the exception of *cd*, all commands are executed by only a single processor, so that files/directories are not being manipulated by multiple processors.

The *cd* style executes the Unix "cd" command to change the working directory. All subsequent SPPARKS commands that read/write files will use the new directory. All processors execute this command.

The *mkdir* style executes the Unix "mkdir" command to create one or more directories.

The *mv* style executes the Unix "mv" command to rename a file and/or move it to a new directory.

The *rm* style executes the Unix "rm" command to remove one or more files.

The *rmdir* style executes the Unix "rmdir" command to remove one or more directories. A directory must be empty to be successfully removed.

Restrictions:

SPPARKS does not detect errors or print warnings when any of these Unix commands execute. E.g. if the specified directory does not exist, executing the *cd* command will silently not do anything.

Related commands: none

Default: none

app_style command

Syntax:

```
app_style style args
```

- style = application style name
- args = args

Examples:

```
app_style ising 100 100
app_style potts 1000 1000 4
```

Description:

This command ...

Restrictions: none

Related commands:

variable, ...

Default: none

app_style command

Syntax:

```
app_style style args
```

- style = application style name
- args = args

Examples:

```
app_style ising 100 100
app_style potts 1000 1000 4
```

Description:

This command ...

Restrictions: none

Related commands:

variable, ...

Default: none

solve_style command

Syntax:

```
solve_style style args keyword value ...
```

- style = *linear* or *tree* or *group*
- *linear* arg = none *tree* arg = none *group* args = hi lo hi,lo = range of allowed probabilities zero or more keyword/value pairs may be appended
- keyword = *ngroup*

```
  ngroup value = N
      N = # of groups to use
```

Examples:

```
solve_style linear
solve_style tree
solve_style group 1.0 1.0e-6
solve_style group 100.0 1.0 ngroup 10
```

Description:

Choose a kinetic Monte Carlo (KMC) solver to use in your application. If no sweeper is used then a solver is required.

A KMC solver picks events for your application to perform from a list of events and their associated probabilities. It does this using the standard Gillespie or BKL algorithm which also computes a timestep during which the chosen event occus. The only difference between the various solver styles is the algorithm they use to select events which affects their speed and scalability as a function of the number of events they choose from. The *linear* solver may be suitable for simulations with few events; the *tree* or *group* solver should be used for larger simulations.

The *linear* style chooses an event by scanning the list of events in a linear fashion. Hence the cost to pick an event scales as O(N), where N is the number of events.

The *tree* style chooses an event by creating a binary tree of probabilities and their sums, as in the Gibson/Bruck implementation of the Gillespie direct method algorithm. Its cost to pick an event scales as O(logN).

The *group* style chooses an event using the composition and rejection (CR) algorithm described originally in Devroye and discussed in Slepoy. Its cost to pick an event scales as O(1) as it is a constant time algorithm. It requires that you bound the *hi* and *lo* probabilities for any event that will be registered with the solver. Note that on−lattice applications typically register the total probability of all a site's events with the KMC solver. The value of *lo* must be > 0.0 and *lo* cannot be >= *hi*.

By default, the *group* style will create groups whose boundaries cascade upward in powers of 2 from *lo* to *hi*. I.e. the first group is from lo to 2*lo, the second group is from 2*lo to 4*lo, etc. Note that for hi/lo = 1.0e6, there would thus be about 20 groups.

If the *ngroup* keyword is used, then it specifies the number of groups to use between *lo* and *hi* and they will be equal in extent. E.g. for *ngroup* = 3, the first group is from lo to lo + (hi−lo)/3, the second group is from lo +

2*(hi−lo)/3, and the third group is from lo + 2*(hi−lo)/3 to hi.

Restrictions:

The *ngroup* keyword can only be used with style *group*.

Related commands:

app_style, sweep_style

Default: none

(Gillespie) Gillespie, J Comp Phys, 22, 403−434 (1976); Gillespie, J Phys Chem, 81, 2340−2361 (1977).

(BKL) Bortz, Kalos, Lebowitz, J Comp Phys, 17, 10 (1975).

(Gibson) Gibson and Bruck, J Phys Chem, 104, 1876 (2000).

(Devroye) Devroye, Non−Uniform Random Variate Generation, Springer−Verlag, New York (1986).

(Slepoy) Slepoy, Thompson, Plimpton, J Chem Phys, 128, 205101 (2008).

app_style command

Syntax:

```
app_style style args
```

- style = application style name
- args = args

Examples:

```
app_style ising 100 100
app_style potts 1000 1000 4
```

Description:

This command ...

Restrictions: none

Related commands:

variable, ...

Default: none

stats command

Syntax:

```
stats delta keyword values ...
```

- delta = time increment between statistical output (seconds)
- zero or more keyword/value pairs may be appended
- keyword = *delay* or *logfreq*

```
  delay value = tdelay
      tdelay = delay stats until at least this time (seconds)
    logfreq values = N factor
      N = number of repetitions per interval
      factor = scale factor between interval
```

Examples:

```
stats 0.1
stats 0.1 delta 0.5
stats 1.0 logfreq 7 10.0
```

Description:

Print statistics to the screen and log file every so many seconds during a simulation. A value of 0.0 for delta means only print stats at the beginning and end of the run, in which case no optional keywords can be used.

The quantities printed are elapsed CPU time followed by those provided by the application, followed by those provided by any diagnostics you have defined.

Typically the application reports only the number of events or sweeps executed, followed by the simulation time, but other application−specific quantities may also be reported. Quantities such as the total energy of the system can be included in the output by creating diagnostics via the diag_style command.

The *delay* keyword will suppress output until *tdelay* time has elapsed.

Using the *logfreq* keyword will produce statistical output at varying intervals during the course of a simulation. There will be *N* outputs per interval where the size of the interval is initially *delta* and then scales up by *factor* each time.

For example, this command

```
stats 0.1 logfreq 7 10.0
```

will produce output at these times:

```
t = 0, 0.1, 0.2, ..., 0.7, 1, 2, ...., 7, 10, 20, ....
```

This command

```
stats 0.1 logfreq 1 2.0
```

will produce output at these times:

```
t = 0, 0.1, 0.2, 0.4, 0.8, 1.6, ...
```

If *N* is specified as 0, then this will turn off logarithmic output, and revert to regular output every *delta* seconds.

Restrictions:

See the doc pages for quantities provided by particular app_style and diag_style commands for further details.

Related commands:

dump, diag_style

Default:

The default stats delta is 0.0. The keyword defaults are delay = 0.0 and no logarithmic output.

sweep command

Syntax:

```
sweep style keyword value ...
```

- style = *random* or *raster* or *color* or *color/strict*
- zero or more keyword/value pairs may be appended
- keyword = *mask*

```
mask value = yes or no
    yes/no = mask out sites than cannot change
```

Examples:

```
sweep random
sweep raster mask yes ...
```

Description:

Use a rejection kinetic Monte Carlo (rKMC) algorithm for an on−lattice application. If rKMC is not used then a kinetic Monte Carlo (KMC) algorithm must be used as defined by the solve_style command.

The rKMC algorithm in SPPARKS selects sites on a lattice in an order determined by this command and requests that the application perform events. The application defines the geometry and connectivity of the lattice, what the possible events are, and defines their rates and acceptance/rejection criteria.

The ordering of selected sites is also affected by the sector command, which partitions each processor's portion of the simulation domain into sectors which are quadrants (2d) or octants (3d). In this case, the ordering described below is within each sector. Sectors are looped over one at a time, interleaved by communication of lattice values inbetween.

For the *random* style, sites are chosen randomly, one at a time.

For the *raster* style, a sweep of the lattice is done, as a loop over all sites in a pre−determined order, e.g. a triple loop over i,j,k for a 3d cubic lattice.

For the *color* style, lattice sites are partitioned into sub−groups or colors which are non−interacting in the sense that events on two sites of the same color can be perfored simultaneously without conflict. This enables parallelism since events on all sites of the same color can be attempted simultaneously. Similar to sectors, the colors are looped over, interleaved by communication of lattice values inbetween.

The *color/strict* style is the same as the *color* style except that random numbers are generated in a way that is independent of the processor which generates them. Thus SPPARKS should produce the same answer, independent of how many processors are used. This can be useful in debugging an application.

If the application supports it, the *mask* keyword can be set to *yes* to skip sites which cannot perform an event due to the current value of the site and its neighbors. Enabling masking should not change the answer given by a simulation (in a statistical sense); it only offers a computational speed−up. For example, sites in the interior of grains in a Potts grain−growth model may have no potential of flipping their value. Masking can only be set to *yes* if the temperature is set to 0.0, since otherwise there is a finite probability of any site performing an event.

Restrictions:

This command can only be used as part of on−lattice applications as specified by the app_style command.

Not all lattice styles and applications support the *color* and *color/strict* styles. Not all applications support the *mask* option.

Related commands:

app_style, solve_style, sector

Default:

The option defaults are mask = no.

temperature command

Syntax:

```
temperature T
```

- T = value of temperature for the Monte Carlo simulation (energy units)

Examples:

```
temperature 2.0
```

Description:

This command sets the temperature as used in various applications. The typical usage would be as part of a Boltzmann factor that alters the propabilities of event acceptance and rejection.

The units of the specfied temperature should be consistent with how the application defines energy. E.g. if used in a Boltzmann factor where a kT factor scales the energy of a Hamiltonian defined by the application, then this command is really defining kT and the specified value should have the units of energy as computed by the Hamiltonian.

Restrictions: none

This command can only be used as part of applications that allow for a temperature to be specified. See the doc pages for individual applications defined by the app_style command for further details.

Related commands: none

Default:

The default temperature is 0.0.

undump command

Syntax:

```
undump dump-ID
```

- dump-ID = ID of previously defined dump

Examples:

```
undump mine
undump 2
```

Description:

Turn off a previously defined dump command so that it is no longer active. This closes the file associated with the dump.

Restrictions: none

Related commands:

dump

Default: none

variable command

Syntax:

```
variable name style args ...
```

- name = name of variable to define
- style = *index* or *loop* or *world* or *universe* or *uloop* or *equal* or *atom*

```
  index args = one or more strings
    loop args = N = integer size of loop
    world args = one string for each partition of processors
    universe args = one or more strings
    uloop args = N = integer size of loop
    equal args = one formula containing numbers, math operations, variable references
      numbers = 0.0, 100, −5.4, 2.8e−4, etc
      math operations = (), −x, x+y, x−y, x*y, x/y, x^y,
                        sqrt(x), exp(x), ln(x), log(x),
                        sin(x), cos(x), tan(x), asin(x), acos(x), atan(x),
                        ceil(x), floor(x), round(x)
      other variables = v_abc, v_n
```

Examples:

```
variable x index run1 run2 run3 run4 run5 run6 run7 run8
variable LoopVar loop $n
variable MyValue equal 5.0*exp(v_energy/(v_boltz*v_Temp))
variable beta equal v_temp/3.0
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
```

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 SPPARKS. 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* can be evaluated to produce a single numeric value which can be output directly via the print command.

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 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 variable setting in the input script.

There are two exceptions to this rule. First, variables of style *equal* ARE redefined each time the command is encountered. This allows them to be reset, when their formulas contain a substitution for another variable, e.g. $x. This can be useful in a loop. This also means an *equal*−style variable will re−define a command−line switch −var setting, so an *index*−style variable should be used for such settings instead, as in bench/in.lj.

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.

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.

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. 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.

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 SPPARKS 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.

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 SPPARKS 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.spparks.variable" and "tmp.spparks.variable.lock" which you will see in your directory during such a SPPARKS 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* style, 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.

Note that *equal* variables can produce different values at different stages of the input script or at different times during a run.

The next command cannot be used with *equal* style variables, since there is only one string.

The formula for an *equal* 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 "2.0 + v_MyTemp / pow(v_Volume,1/3)"
```

Specifically, an formula can contain numbers, math operations, and references to other variables.

Number	0.2, 100, 1.0e20, −15.4, etc
Math operations	(), −x, x+y, x−y, x*y, x/y, x^y, sqrt(x), exp(x), ln(x), log(x), sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), ceil(x), floor(x), round(x)
Other variables	v_abc, v_n

Math operations are written in the usual way, where the "x" and "y" in the examples above can be another section of the formula. Operators are evaluated left to right and have the usual precedence: unary minus before exponentiation ("^"), exponentiation before multiplication and division, and multiplication and division before addition and subtraction. Parenthesis can be used to group one or more portions of a formula and enforce a desired order of operations. Additional math operations can be specified as keywords followed by a parenthesized argument, e.g. sqrt(v_ke). Note that ln() is the natural log; log() is the base 10 log. The ceil(), floor(), and round() operations 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.

The current values of other variables can be accessed by prepending a "v_" to the variable name. This will cause that variable to be evaluated.

IMPORTANT NOTE: If you define variables in circular manner like this:

```
variable a equal v_b
variable b equal v_a
print $a
```

then SPPARKS will run for a while when the print statement is invoked!

Another way to reference a variable in a formula is using the $x form instead of v_x. There is a subtle difference between the two references that has to do with when the evaluation of the included variable is done.

Using a $x, the value of the include variable is substituted for immediately when the line is read from the input script, just as it would be in other input script command. This could be the desired behavior if a static value is desired. Or it could be the desired behavior for an equal−style variable if the variable command appears in a loop (see the jump and next commands), since the substitution will be performed anew each time thru the loop as the command is re−read. Note that if the variable formula is enclosed in double quotes, this prevents variable substitution and thus an error will be generated when the variable formula is evaluated.

Using a v_x, the value of the included variable will not be accessed until the variable formula is evaluated. Thus the value may change each time the evaluation is performed. This may also be desired behavior.

As an example, if the current simulation box volume is 1000.0, then these lines:

```
variable x equal vol
variable y equal 2*$x
```

will associate the equation string "2*1000.0" with variable y.

By contrast, these lines:

```
variable x equal vol
variable y equal 2*v_x
```

will associate the equation string "2*v_x" with variable y.

Thus if the variable y were evaluated periodically during a run where the box volume changed, the resulting value would always be 2000.0 for the first case, but would change dynamically for the second case.

Restrictions:

All *universe*− and *uloop*−style variables defined in an input script must have the same number of values.

Related commands:

next, jump, include, print

Default: none

volume command

Syntax:

```
volume V
```

- V = volume of system (liters)

Examples:

```
volume 1.0e-10
```

Description:

This command sets the volume of the system for use in the app_style chemistry application.

For example, it could be the volume of a biological cell within which biochemical reactions are taking place.

Restrictions:

This command can only be used as part of the app_style chemistry application.

Related commands:

app_style chemistry

Default: none