SPPARKS Users Manual

Stochastic Parallel PARticle Kinetic Simulator

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

Table of Contents

Table of Contents

SPPARKS Documentation

(23 July 2008 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 MC 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 kinetic Monte Carlo (KMC) code that has algorithms for both KMC and Metropolis MC updating. In a generic sense its KMC 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 indivudually at random or a sweep of a geometric lattice can be performed to select possible events in an ordered fashion. In the Metropolis context an event may be accepted or rejected via a Boltzmann criterion.

Various applications are implemented in SPPARKS which use this solver/sweeper framework to perform simulations as a series of events. The application defines the events and their probabilities. It also executes each event when it is selected.

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

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 the applications currently available in SPPARKS:

- Ising model
- Potts model
- Membrane model
- Biochemcial reaction network model

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

- linear, O(N)
- tree, O(logN)
- group, 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.

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.

2.3 Making SPPARKS with optional packages

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".

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

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, sweeper

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

(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, solve_style, sweep_style

Application−specific commands:

add_reaction, add_species, count, inclusion, temperature, volume

Output commands:

diag_style, dump, stats

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	clear	count	diag_style
dump	echo	if	include	inclusion	jump

label	log	next	print	run	shell
solve_style	stats	sweep_style	temperature	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	ising	membrane	potts	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

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

lattice	lattice2d	lattice3d

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	energy	eprof3d

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, add 2 lines to the style_user.h file, and re−make the code.

The first line added to style_user.h would be

```
AppStyle(foo,AppFoo)
```

in the #ifdef AppClass section, where "foo" is the style keyword in the app_style command, and AppFoo is the class name in your C++ files.

The 2nd line added to style_user.h would be

```
#include "app_foo.h"
```

in the #ifdef AppInclude section, where app_foo.h is the name of your new include file.

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
- Sweep 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 should derive from the AppLattice (general lattice), AppLattice2d (2d square lattice), of AppLattice3d (3d cubic lattice) classes. The 2d and 3d variants may disappear in the future as they are superceded by AppLattice, so that style of application should be your default choice.

For 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	additional commands the application defines
init	setup the application
run	perform iterations or timestepping of the model
dump_header	write header of dump file
dump	write a snapshot of state of model
set_stats	setup application−specific statistics
set_dump	setup application−specific dump

For 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_lattice.h for details.

site_energy	compute energy of a lattice site
site_event_rejection	peform a Metropolis event with rejection
site_propensity	compute propensity of all events on a site
site_event	perform a kinetic Monte Carlo event
input_app	perform application−specific

	input
init_app	perform application−specific initialization

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

Sweep styles

In SPPARKS, a sweep performs Metropolis Monte Carlo or kinetic Monte Carlo operations on a series of lattice sites for on−lattice applications. See the sweep_style command for details. See the app_style command for info on on−lattice applications.

There are 3 sweep classes that support the various kinds of lattices used by applications: SweepLattice for general lattices, SweepLattice2d for 2d square lattices, and SweepLattice3d for 3d cubic lattices.

Each of these sweep classes has lo−level functions defined for various kinds of sweeping: vanilla sweeping, with masking, with coloring for strict sweeping, etc. New functions could be added to any of the these 3 classes for alternate sweeping strategies that could then be called by the top−level of the sweeper.

See the sweep_lattice.h, sweep_lattice2d.h, and sweep_lattice3d.h files for details.

(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:

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.

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.

Cannot mask sweeping with non−zero temperature
: A finite temperature implies random spin flips can occur. Thus a site cannot be masked out with 100% certainty.

Cannot open diag style cluster3d dump file
: Self−explanatory.

Cannot open diag_style cluster dump file
: Self−explanatory.

Cannot open diag_style cluster output file
: Self−explanatory.

Cannot open diag_style cluster2d dump file
: Self−explanatory.

Cannot open diag_style cluster2d output file
: Self−explanatory.

Cannot open diag_style cluster3d dump file
: Self−explanatory.

Cannot open diag_style cluster3d output file
: Self−explanatory.

Cannot open diag_style energy output file

Self−explanatory.

Cannot open diag_style energy2d output file

Self−explanatory.

Cannot open diag_style energy3d output file

Self−explanatory.

Cannot open diag_style eprof3d output file

Self−explanatory.

Cannot open dump file

Self−explanatory.

Cannot open file %s

Self−explanatory.

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 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 use solver in parallel

A KMC solver cannot be used in parallel without a sweep style being defined.

Cannot use solver with non−KMC sweeper

Can only use a KMC solver with a sweep style that invokes the KMC option.

Command used before app_style set

A command is assumed to be application−specific, but is used before the app_style command defines the application.

Connectivity not defined for this AppLattice child class

Cannot use a diagnostic that requires connectivity for an application derived from AppLattice2d or AppLattice3d.

Delevent > delpropensity

Such an application does not make sense.

Diag style cluster3d 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.

Divide by 0 in variable formula

Self−explanatory.

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.

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 combination of sweep flags
Self−explanatory.

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

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

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 site specification in app_style potts/variable
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.

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

Lattice app needs a solver or sweeper
Self−explanatory.

Lattice per proc is too small
The section of lattice stored by a processor must be large enough to be split into sectors and not overlap too far into other processor's sub−domains.

Log of zero/negative in variable formula

Self−explanatory.

Maxbuftmp size too small in AppGrain::dump_detailed()

Self−explanatory.

Maxbuftmp size too small in AppGrain::dump_detailed_mask()

Self−explanatory.

Maxbuftmp size too small in DiagCluster2d::dump_clusters()

Self−explanatory.

Maxbuftmp size too small in DiagCluster3d::dump_clusters()

Self−explanatory.

Maxbuftmp size too small in DiagEprof3d::write_prof()

Self−explanatory.

Mismatch in counting for dbufclust

Self−explanatory.

Mismatched sweeper with app lattice

The lattice styles must match between the sweeper and application.

Must define solver with KMC sweeper

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.

No reactions defined for chemistry app

Use the add_reaction command to specify one or more reactions.

No solver class defined

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.

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.

Site−site interaction was not found

Internal SPPARKS error. Should not occur.

Species ID %s already exists

Self−explanatory.

Species ID %s does not exist

Self−explanatory.

Sqrt of negative in variable formula

Self−explanatory.

Substitution for undefined variable

Self−explanatory.

Sweep option not yet supported

Not all sweep options are currently supported with all lattice styles.

Unbalanced quotes in input line

No matching end double quote was found following a leading double quote.

Unexpected end of lattice file

Self−explanatory.

Unexpected end of lattice spin file

Self−explanatory.

Unexpected value in spin 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 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.

Variable name must be alphanumeric or underscore characters

Self−explanatory.

Vertices read from file incorrectly

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 application 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
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 ising command

app_style ising/exchange command

app_style ising/2d/4n command

app_style ising/2d/4n/exchange command

app_style ising/2d/8n command

app_style ising/3d/6n command

app_style ising/3d/26n command

Syntax:

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

- style = *ising* or *ising/exchange* or *ising/2d/4n* or *ising/2d/4n/exchange* or *ising/2d/8n* or *ising/3d/6n* or *ising/3d/26n*

```
ising arg = seed
  ising/exchange arg = seed
    seed = random number seed (positive integer)
  ising/2d/4n args = Nx Ny seed
  ising/2d/4n/exchange args = Nx Ny seed
  ising/2d/8n args = Nx Ny seed
  ising/3d/6n args = Nx Ny Nz seed
  ising/3d/26n args = Nx Ny Nz seed
    Nx,Ny = size of 2d lattice size
    Nx,Ny,Nz = size of 3d lattice size
    seed = random number seed (positive integer)
```

- see the app_style command for additional keywords that can be appended to the *ising* and *ising/exchange* styles

Examples:

```
app_style ising 18874 lattice sq/4n 1.0 50 50
app_style ising/exchange 18874 lattice tri 1.0 50 50
app_style ising/2d/4n 50 50 18874
app_style ising/2d/4n/exchange 50 50 18874
app_style ising/2d/8n 100 100 48783
app_style ising/3d/6n 100 100 100 887287
app_style ising/3d/26n 50 50 100 487827
```

Description:

These applications 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:

```
H = − Sum_j delta_ij
```

where Sum_j is a sum over all the neighbor sites of site I and delta_ij is 1 if the spin of sites I and J are the same and 0 otherwise.

These Ising models can be run in one of several modes, either as a general lattice application or as a 2d square lattice application or as a 3d square lattice application. See the app_style commmand for further discussion. When running as a general lattice application, the lattice is specified by the appended *lattice* keword with its associated values, as discussed on the doc page for the app_style command. When run as a 2d or 3d lattice application the style name also determines how many neighbors per site are defined:

- 2d/4n = 2d square lattice with 4 neighbors per site (nearest neighbors)
- 2d/8n = 2d square lattice with 8 neighbors per site (1st and 2nd nearest neighbors)
- 3d/6n = 3d cubic lattice with 6 neighbors per site (nearest neighbors)
- 3d/26n = 3d cubic lattice with 26 neighbors per site (1st,2nd,3rd nearest neighbors)

All the Ising applications except those with "exchange" in the style name perform Glauber dynamics, meaning they flip the spin on a single site. The applications with "exchange" in the style name perform 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 Metropolis rejection−based algorithm. You must thus define a sweeping method and/or KMC solver to be used with the application via the sweep_style and solve_style commands.

For the Glauber dynamics applications with 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 the Glauber dynamics applications with solution by a Metropolis algorithm, the spin is set randomly to 1 or 2 and 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)].

For the Kawasaki dynamics applications with solution by a KMC algorithm, the possible events a site can perform are swaps with any neighbor site with a spin different than itself. The probability of each such event is min[1,exp(−dE/kT)], where dE = Efinal − Einitial and E is the sum of the energy for the site and its neighbor.

For the Kawasaki dynamics applications with solution by a Metropolis algorithm, the spin is flipped to its opposite state and dE = Efinal − Einitial is calculated, as is a uniform random number R between 0 and 1. The flip is accepted if R < min[1,exp(−dE/kT)], else it is rejected.

There are no additional commands defined by these applications.

Restrictions: none

Related commands:

app_style potts

Default: none

app_style membrane command

app_style membrane/2d command

Syntax:

```
app_style style args
```

- style = *membrane* or *membrane/2d*

```
  membrane args = w01 w11 mu seed
      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)
      seed = random number seed (positive integer)
    membrane/2d args = Nx Ny w01 w11 mu seed
      Nx,Ny = size of 2d square lattice with 4 neighbors per site
      w01,w11,mu,seed = same as above
```

- 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 12345 lattice tri 1.0 100 50
app_style membrane/2d 100 100 1.25 1.0 −3.0 12345
```

Description:

These applications evolve 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.

These membrane models can be run in one of 2 modes, either as a general lattice application or as a 2d square lattice application with 4 neighbors per site. See the app_style commmand for further discussion. When running as a general lattice application, the lattice is specified by the appended *lattice* keword with its associated values, as discussed on the doc page for the app_style command.

As explained on this page, these applications can be evolved by either a kinetic Monte Carlo (KMC) or Metropolis rejection−based algorithm. You must thus define a sweeping method and/or KMC solver to be used with the application via the sweep_style and solve_style 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 command is defined by these applications:

inclusion	specify which sites are proteins

Restrictions: none

Related commands: none

Default: none

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

app_style potts command

app_style potts/variable command

app_style potts/2d/4n command

app_style potts/2d/8n command

app_style potts/2d/24n command

app_style potts/3d/6n command

app_style potts/3d/12n command

app_style potts/3d/26n command

Syntax:

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

- style = *potts* or *potts/variable* or *potts/2d/4n* or *potts/2d/8n* or *potts/2d/24n* or *potts/3d/6n* or *potts/3d/12n* or *potts/3d/26n*

```
potts arg = Q seed
  potts/variable arg = Q seed
    Q = number of spin states
    seed = random number seed (positive integer)
  potts/2d/4n args = Nx Ny Q seed
  potts/2d/8n args = Nx Ny Q seed
  potts/2d/24n args = Nx Ny Q seed
  potts/3d/6n args = Nx Ny Nz Q seed
  potts/3d/12n args = Nx Ny Nz Q seed
  potts/3d/26n args = Nx Ny Nz Q seed
    Nx,Ny = size of 2d lattice size
    Nx,Ny,Nz = size of 3d lattice size
    Q = number of spin states
    seed = random number seed (positive integer)
```

- see the app_style command for additional keywords that can be appended to the *potts* and *potts/variable* styles

Examples:

```
app_style potts 20 18874 lattice sq/4n 1.0 50 50
app_style potts/variable 20 18874 lattice sq/4n 1.0 50 50 site 1 1
app_style potts/2d/4n 50 50 20 18874
app_style potts/2d/8n 100 100 20 48783
app_style potts/2d/24n 100 100 20 48783
app_style potts/3d/6n 100 100 100 100 887287
app_style potts/3d/12n 100 100 100 100 887287
app_style potts/3d/26n 50 50 100 1000 487827
```

Description:

These applications evolve a Q−state Ising 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:

```
H = − Sum_j delta_ij
```

where Sum_j is a sum over all the neighbor sites of site I and delta_ij is 1 if the spin of sites I and J are the same and 0 otherwise.

These Potts models can be run in one of several modes, either as a general lattice application or as a 2d square lattice application or as a 3d square lattice application. See the app_style commmand for further discussion. When running as a general lattice application, the lattice is specified by the appended *lattice* keword with its associated values, as discussed on the doc page for the app_style command. When run as a 2d or 3d lattice application the style name also determines how many neighbors per site are defined:

- 2d/4n = 2d square lattice with 4 neighbors per site (nearest neighbors)
- 2d/8n = 2d square lattice with 8 neighbors per site (1st and 2nd nearest neighbors)
- 2d/24n = 2d square lattice with 24 neighbors per site (5x5 stencil surrounding each site)
- 3d/6n = 3d cubic lattice with 6 neighbors per site (nearest neighbors)
- 3d/12n = 3d cubic lattice with 12 neighbors per site (2 neighbors in each of 6 directions)
- 3d/26n = 3d cubic lattice with 26 neighbors per site (1st,2nd,3rd nearest neighbors)

All the Potts applications perform Glauber dynamics, meaning they flip the spin on a single site. The *potts/variable* application is identical to the *potts* application except that the *site* keyword (described with the app_style command) can be used to define multiple quantities per site, instead of just a single spin value.

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

For solution by a KMC algorithm, a site event is a spin flip to a spin value of a neighbor site different than itself. For example, let a site have 12 neighbors and those 12 spins have 4 different values, one of which is the same as the central site. There will then be 3 events defined for the central site. The probability of each 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 spin is set to a random value between 1 and Q, 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.

There are no additional commands defined by these applications.

Restrictions: none

Related commands:

app_style ising

Default: none

app_style command

Syntax:

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

- style = *ising* or *potts* or *membrane* or *chemistry* or *test/group*

  (see application doc page for additional variants)
- args = arguments specific to an application

  (see application doc page for details)
- zero or more keyword/value pairs may be appended
- keyword = *lattice* or *site*

```
  lattice values = type params
      type = sq/4n or sq/8n or tri or sc/6n or sc/26n or fcc or bcc or diamond or random/2d or r
        sq/4n params = a nx ny = 2d square lattice with 4 neighbors per site
        sq/8n params = a nx ny = 2d square lattice with 8 neighbors per site
        tri params = a nx ny = 2d triangular lattice with 6 neighborrs per site
          a = lattice constant
          nx,ny = number of unit cells in each dimension
        sc/6n params = a nx ny nz = 3d cubic lattice with 6 neighbors per site
        sc/26n params = a nx ny nz = 3d cubic lattice with 26 neighbors per site
        fcc params = a nx ny nz = 3d fcc lattice with 12 neighbors per site
        bcc params = a nx ny nz = 3d bcc lattice with 8 neighbors per site
        diamond params = a nx ny nz = 3d diamond lattice with 4 neighbors per site
          a = lattice constant
          nx,ny,nz = number of unit cells in each dimension
        random/2d param = N xbox ybox cutoff = lattice of random 2d points
          N = # of lattice points
          xbox,ybox = simulation extent in x,y
          cutoff = distance cutoff for neighbor connectivity between sites
          seed = random number seed (positive integer)
        random/3d param = N xbox ybox zbox cutoff = lattice of random 3d points
          N = # of lattice points
          xbox,ybox,zbox = simulation extent in x,y,z
          cutoff = distance cutoff for neighbor connectivity between sites
          seed = random number seed (positive integer)
        file param = filename = read lattice and connectivity from file
          filename = name of file (see file format below)
    site values = Nint Ndouble
      Nint = # of integer quantites to store per site
      Ndouble = # of double quantites to store per site
```

Examples:

```
app_style ising ... lattice sq/4n 1.0 100 100
app_style ising/exchange ... lattice fcc 1.0 100 100 100
app_style ising/2d/4n ...
app_style ising/2d/4n/exchange ...
app_style ising/2d/8n ...
app_style ising/3d/6n ...
app_style ising/3d/26n ...

app_style potts ... lattice file tmp.latttice
app_style potts/variable ... lattice random/2d 1000 10.0 10.0 3.0 49893
app_style potts/2d/4n ...
```

```
app_style potts/2d/8n ...
app_style potts/2d/24n ...
app_style potts/3d/6n ...
app_style potts/3d/12n ...
app_style potts/3d/26n ...

app_style membrane ... lattice tri 1.0 100 50
app_style membrane/2d ...

app_style chemistry ...

app_style test/group ...
```

Description:

This command defines what model or application SPPARKS will run. There are 2 basic kinds of applications: on−lattice and off−lattice. For lattice−based simulations, there are 3 variants: those on a 2d square lattice, on a 3d cubic lattice, and on a general lattice which can be either 2d or 3d and of various kinds. The reason for these variants is that we are still experimenting with the best way to write lattice appications in SPPARKS. Eventually, we will probably support only the general lattice option, so it is probably best to develop new on−lattice MC applications for that style.

Here is the list of on−lattice applications SPPARKS currently includes. Each of these applications has 2d, 3d, and general lattice variants as illustrated above. See the doc for each application for details:

- ising = Ising model
- membrane = membrane model of lipid,water,protein
- potts = multi−state Potts model for grain growth

Here is the list of off−lattice applications SPPARKS currently includes:

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

Depending on the application, the model it represents can be evolved in one or two ways.

The first method is to use a kinetic Monte Carlo (KMC) solver, specified by the solve_style command. If the application supports parallel KMC evolution, then the "sweep_style kmc yes" command is also used in conjunction with a KMC solver. In this scenario the application defines a list of "events" and associated probabilities, the solver chooses the next event, and the application updates the system accordingly. For off−lattice applications the definition of an "event" is arbitrary. For on−lattice application zero or more possible events are typically defined for each lattice site.

The second method, which is supported by many of the on−lattice applications, is to sweep over the lattice and use a rejection−based Metropolis algorithm to accept or reject events occuring on each site. The style of sweeping is specified by the sweep_style command. No solve_style command is specified in this scenario.

For both methods (KMC and Metropolis) the rules for how events are chosen and accepted or rejected are discussed in the doc pages for the individual applications.

For 2d and 3d lattice applications, the kind of lattice used and its size are determined by the application style and its arguments. E.g.

```
app_style ising/2d/4n 100 100 12345
```

means a 2d square lattice of size 100x100 with 4 neighbors per lattice. The only lattice types supported in this mode are 2d square and 3d cubic, though the neighbor stencil can be of various kinds (e.g. 4 or 8 neighbors in 2d). See the doc pages for individual applications for details.

For general lattice applications, the kind of lattice used and its size are independent of the application and must be specified by the *lattice* keyword and its values. E.g.

```
app_style ising 12345 lattice sq/4n 100 100
```

means exactly the same thing as the 2d lattice example above, namely a 2d square lattice of size 100x100 with 4 neighbors per lattice site. A variety of lattice types and neighbor stencils are given as options with the *lattice* keyword as described above.

The *sq* and *sc* lattice types are 2d square and 3d cubic lattices The total number of lattice sites is one per unit cell, i.e. the product of *nx*, *ny*, and *nz*. The *fcc*, *bcc*, and *diamond* lattice types are 3d and generate multiple lattice sites per unit cell: 4 per fcc unit cell, 2 per bcc unit cell, and 8 per diamond unit cell.

The connectivity of these lattice types is as follows:

- sq/4n = 2d square lattice with 4 neighbors per site (nearest neighbors)
- sq/8n = 2d square lattice with 4 neighbors per site (1st and 2nd nearest neighbors)
- tri = 2d triangular lattice with 6 neighbors per site (nearest neighbors)
- sc/6n = 3d cubic lattice with 6 neighbors per site (nearest neighbors)
- sc/26n = 3d cubic lattice with 26 neighbors per site (1st,2nd,3rd nearest neighbors)
- fcc = 3d fcc lattice with 12 neighbors per site (nearest neighbors)
- bcc = 3d fcc lattice with 8 neighbors per site (nearest neighbors)
- diamond = 3d fcc lattice with 4 neighbors per site (nearest neighbors)

The *random/2d* and *random/3d* lattice options generate a lattice of random points within a 2d or 3d box of specified size (0−xbox,0−ybox,0−zbox), generated by a random number generator using the specified *seed*. The *cutoff* criterion is used to assign lattice neighbors to each site.

The *file* lattice option reads in a lattice and neighbor connectivity from the specified *filename*. The format of this file is as follows where the comments (#) are not included in the file, and "vertex" is a lattice site, and an "edge" is a neighbor connection from one site to another. Typically neighbors should be geometrically close, but that is not required. Note that a connection between two sites is listed twice, once as edge IJ, and once as edge JI.

```
comment              # 1st line is skipped
                     # skipped line
Ndim dimension       # Ndim = 2 or 3
N vertices           # N = number of vertices
M max connectivity   # M = maximum number of edges for any vertex
X1 X2 xlo xhi        # X1,X2 = x bounds of box that encloses lattice
Y1 Y2 xlo xhi        # y bounds
Z1 Z2 zlo zhi        # z bounds (only if Ndim = 3)
                     # skipped line
Vertices
                     # skipped line
1 x y z              # ID, x, y, z for each vertex
2 x y z              # no z value if dim = 2
...
N x y z              # N lines in this section
                     # skipped line
Edges
                     # skipped line
1 n1 n2 n3 ...       # ID, list of IDs for neighbor connections
```

```
1 n1 n2 n3 ...       # can be different number of connections (up to M) for each vertex
...
N n1 n2 n3 ...       # N lines in this section
```

For 2d and 3d lattice applications, there is always one integer stored per lattice site, which your application can access and update.

For general lattice applications, the *site* keyword can be used. By default each lattice site stores a single integer value. By specifying *site*, multiple integer and or double values can be stored on each site and accessed/updated by your application. For example, an integer flag could be stored for the type of lattice site and one or more doubles could store the state of the site. If *site 0 0* is specified, then the default of a single integer per site is used.

Restrictions: none

Related commands: none

Default:

There is no default for the *lattice* keyword. It must be specified for on−lattice applications of the general lattice style. The default value for site is 0 0.

app_style test/group command

Syntax:

```
app_style test/group N Nmax pmax pmin delta seed 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
- seed = random number seed (positive integer) 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 49289
app_style test/group 10000 30 1.0 1.0e-9 10 5982893 lomem yes
```

Description:

This application 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

There are no additional commands defined by this application.

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.

Restrictions: none

Related commands:

solve_style group

Default:

The default value is lomem = no.

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

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.

diag_style cluster command

diag_style cluster2d command

diag_style cluster3d command

app_style command

Syntax:

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

- style = *cluster* or *cluster2d* or *cluster3d*
- delta = time increment between evaluations of the diagnostic (seconds)
- zero or more keyword/value pairs may be appended
- 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
```

- see the diag_style command for additional keywords that can be appended to any diagnostic command

Examples:

```
diag_style cluster 0.5
diag_style cluster2d 0.5
diag_style cluster3d 0.5
diag_style cluster3d 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 *filename* keyword allows a file to be specified which a full listing of cluster statistics is written to.

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 output with two columns: site index and cluster ID. It can be visualized with various tools in the LAMMPS package and the Pizza.py package. The *opendx* setting generates a file that can be read by the OpenDX script called aniso0.net to visualize the clusters in 3D. OpenDX can also be used to visualize 2D lattices of site values.

Restrictions:

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.

As described by the app_style command, on−lattice applications use one of 3 styles of lattice: general, 2d, or 3d. For this diagnostic you must use a style that matches the application's lattice. The *cluster* style is for general lattices, the *cluster2d* and *cluster3d* styles are for 2d and 3d lattices.

Related commands:

diag_style stats

Default: none

diag_style energy command

diag_style energy2d command

diag_style energy3d command

Syntax:

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

- style = *energy* or *energy2d* or *energy3d*
- delta = time increment between evaluations of the diagnostic (seconds)
- see the diag_style command for additional keywords that can be appended to any diagnostic command

Examples:

```
diag_style energy 0.5
diag_style energy2d 0.5
diag_style energy3d 0.5
```

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:

As described by the app_style command, on−lattice applications use one of 3 styles of lattice: general, 2d, or 3d. For this diagnostic you must use a style that matches the application's lattice. The *energy* style is for general lattices, the *energy2d* and *energy3d* styles are for 2d and 3d lattices.

Related commands:

diag_style stats

Default: none

diag_style eprof3d command

Syntax:

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

- eprof3d = style name of this diagnostic
- delta = time increment between evaluations of the diagnostic (seconds)
- zero or more keyword/value pairs may be appended
- 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
```

- see the diag_style command for additional keywords that can be appended to any diagnostic command

Examples:

```
diag_style eprof3d 0.1 axis x filename eprof3d.dat
diag_style eprof3d 0.1 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 command

Syntax:

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

- style = *cluster* or *cluster2d* or *cluster3d* or *eprof3d* or *energy* or *energy2d* or *energy3d*
- delta = time increment between evaluations of the diagnostic (seconds)
- zero or more keyword/value pairs may be appended
- keyword = *log*

```
log values = N factor
    N = number of repetitions per interval
    factor = scale factor between interval
```

- see doc pages for individual diagnostic commands for additional keywords that can be appended

Examples:

```
diag_style cluster 1.0
diag_style eprof3d 10.0
diag_style energy2d 0.01 log 7 10.0
```

Description:

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

As described by the app_style command, on−lattice applications use one of 3 styles of lattice: general, 2d, or 3d. For several of the diagnostic styles, you must use a style that matches the application's lattice.

The diagnostics currently available are:

- cluster = grain size statistics for general lattices
- cluster2d = grain size statistics for 2d lattices
- cluster3d = grain size statistics for 3d lattices
- eprof3d = site energy as function of distance from sector boundary for 3d lattices
- energy = compute energy of entire system for general lattices
- energy2d = compute energy of entire system for 2d lattices
- energy3d = compute energy of entire system for 3d lattiecs

All diagnostics provide generate or more values that are appended to other statistical output and printed to the screen and log file via the stats command. We call this stats output. In addition, diagnostic may optionally write more extensive output to dedicated files that are specified using diagnostic−specific keywords.

Using the *log* 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 scales up by *factor* each time. *Delta* is the time between outputs in the first (smallest) interval.

For example, this command

```
diag_style energy 0.01 log 7 10.0
```

will perform its computation at these times:

```
t = 0, 0.01, 0.02, ..., 0.07, 1.0, 2.0, ...., 7.0, 10.0, 20.0, ....
```

This command

```
diag_style energy 0.01 log 1 2.0
```

will perform its computation at these times:

```
t = 0, 0.01, 0.02, 0.04, 0.08, ...
```

Restrictions: none

Related commands:

stats

Default: none

dump command

Syntax:

```
dump delta style filename
```

- delta = time increment between dumps (seconds)
- style = *lattice* or *coord*
- filename = name of file to dump snapshots to

Examples:

```
dump 0.25 lattice tmp.dump
dump 5.0 coord snap.ising
```

Description:

Dump snapshots of the state of the system to a file every so many seconds during a simulation. The quantities printed are obtained from the application. Typically these are the values of the various sites in the simulation.

Two styles of dump file are supported, though not every application supports both. The *lattice* style simply outputs a list of site IDs and values (e.g. spin). The *coord* style outputs a LAMMPS−compatible dump file where each site has a type (e.g. spin value) and an x,y,z coordinate. The latter can be visualized with various tools in the LAMMPS package and the Pizza.py package.

Restrictions: none

Related commands:

stats

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
```

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 for use in the app_style membrane application.

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

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

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

print command

Syntax:

```
print string
```

- string = text string to print. may contain variables

Examples:

```
print "Done with equilibration"
print "The system volume is now $v"
```

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), there are 3 options. First, consider using the fix print command, which will print a string periodically during a simulation. Second, the print command can be used as an argument to the *every* option of the run command. Third, the print command could appear in a section of the input script that is looped over (see the jump and next commands).

See the variable command for a description of *equal* style variables which are typically the most useful ones to use with the print command. Equal−style variables can calculate formulas involving mathematical operations, atom properties, group properties, thermodynamic properties, global values calculated by a compute or fix, or references to other variables.

Restrictions: none

Related commands:

fix print, variable

Default: none

run command

Syntax:

```
run delta
```

- delta = run simulation for this amount of time (seconds)

Examples:

```
run 100.0
run 10000.0
```

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.

Restrictions: none

Related commands: none

Default: none

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 = seed
      seed = random number seed (positive integer)
    tree arg = seed
      seed = random number seed (positive integer)
    group args = hi lo seed
      hi,lo = range of allowed probabilities
      seed = random number seed (positive integer)
```

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

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

Examples:

```
solve_style linear 394893
solve_style tree 1995893
solve_style group 1.0 1.0e-6 6998729
solve_style group 100.0 1.0 59983 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. If a sweeper with the KMC option is used, then a solver is also 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

(Gillepsie) Gillespie, J Chem 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 = *log*

```
  log values = N factor
      N = number of repetitions per interval
      factor = scale factor between interval
```

Examples:

```
stats 0.1
stats 1.0 log 7 10.0
```

Description:

Print statistics to the screen and log file every so many seconds during a simulation. The quantities printed are obtained by calling the stats() functions belongs to the application and each of the diagnostics. 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. Standard quantities such as energy can be included in the statistical output by creating diagnostics via the diag_style command.

Using the *log* 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 scales up by *factor* each time. *Delta* is the time between outputs in the first (smallest) interval.

For example, this command

```
stats 0.01 log 7 10.0
```

will produce output at these times:

```
t = 0, 0.01, 0.02, ..., 0.07, 1.0, 2.0, ...., 7.0, 10.0, 20.0, ....
```

This command

```
stats 0.01 log 1 2.0
```

will produce output at these times:

```
t = 0, 0.01, 0.02, 0.04, 0.08, ...
```

Restrictions:

If stats is called on a timestep when a diagnostic is not called, then the information provided by the diagnostic may be old.

Related commands:

dump

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

sweep_style command

Syntax:

```
sweep_style style seed keyword value keyword value ...
```

- style = *lattice* or *lattice2d* or *lattice3d*
- seed = random number seed (positive integer)
- zero or more keyword/value pairs may be appended

```
keyword = mask or strict or kmc or delt or adapt
  mask value = yes or no
    yes/no = mask out sites than cannot change
  strict value = yes or no
    yes/no = loop over sites in color sets for answers independent of parallelism and masking
  kmc value = yes or no
    yes/no = perform approximate kinetic MC algorithm
  adapt value = yes or no
    yes/no = adapt timestep as the simulation proceeds to preserve accuracy
  delt value = dt
    dt = time associated with one sweep (seconds)
  deln value = factor
    factor = unitless scale factor for time of one sweep scaled by propensity
```

Examples:

```
sweep_style lattice2d seed option arg option arg ...
sweep_style lattice3d seed option arg option arg ...
```

Description:

Choose a sweeping algorithm to use in an on−lattice Monte Carlo application. If no sweeper is used then a kinetic Monte Carlo (KMC) solver is required. If a sweeper with the KMC option is used, then a solver is also required.

As explained in the app_style command, there are 3 kinds of lattices: general, 2d, and 3d. You must use a sweep style that matches the application. Namely, style *lattice* for applications which use general lattices, style *lattice2d* for applications which use 2d lattices, and style *lattice3d* for applications which use 3d lattices.

A sweeping algorithm loops over the sites on a lattice 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 probabilities and acceptance/rejection criteria.

When SPPARKS runs in parallel, the simulation domain is partitioned into sub−domains, one per processor. For 2d lattices, the processors are mapped to a 2d grid of rectangular sub−domains. For 3d lattices, the processors are likewise mapped to a 3d grid of brick−shaped sub−domains. To insure events occurring on one processor do not conflict with events performed by another processor, each sub−domain is partitioned into sectors. There are 4 sectors per sub−domain in 2d, 8 sectors in 3d. Even when running on one processor, the entire domain is partitioned into sectors.

A sweep is performed in the following manner. All processors sweep over sites in the same sector (e.g. their lower left sector in 2d) at the same time. Communication between processors is then performed to update sites on the sector boundary. Then all proecessors move to the next sector, and the process is repeated. Thus a single sweep over the entire lattice is performed in 4 (or 8) stages for 2d (or 3d) lattices, as sectors are swept over one at

a time, followed by the appropriate communication.

There are two basic ways to perform a sweep: ordered and random.

If the *kmc* keyword is *no*, then ordered sweeps are performed. A sector sweep is simply a loop over all sites in the sector. For 2d and 3d lattices the looping is done in a standard way (i,j,k). For general lattices the ordering of the sites may be less structured, particularly for random lattices or lattices read from files. Each time a site is visited, the application is asked to perform an event for the site via the Metropolis Monte Carlo algorithm with an acceptance and rejection criterion.

For ordered sweeps the *mask* and *strict* keywords can also be used separately or together. The *delt* keyword specifies how much time elapses in the simulation per sweep.

if the *mask* keyword is *yes*, the sweep ordering is the same, but sites can be masked if no event is possible to save computational time. For example, in a grain growth model, sites in the interior of a grain may never change. It is up to the application to provide criteria for when a lattice site mask is set or unset. Masking can not be invoked when the temperature is non−zero.

For ordered sweeps, if the *strict* keyword is *yes*, then the lattice sites are partitioned into "colors" such that lattice sites of the same color do not interact with each other in an energetic sense. Extra looping is done over colors and sectors to sweep through all the sites. The purpose of the *strict* option is to produce the same answer independent of the number of processors being used and whether masking is on or off. This can be useful in debugging an application.

If the *kmc* keyword is *yes*, then random sweeps are performed. The *mask* and *strict* keywords cannot be used. In a random "sweep" the sites within one sector are assigned event probabilities by the application. A kinetic Monte Carlo (KMC) algorithm is used to randomly choose which site invokes the next event. Thus a solver style must be specified using the solve_style command. The KMC algorithm also computes a time associated with the selected event and the time for that sector is incremented. Event selection continues until *delt* time has elapsed. Then communication is performed and the next sector executes a series of events. Thus after sweeping within all sectors is completed, *delt* has elapsed for the entire system.

Note that this is really an approximate KMC algorithm, 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 *delt* is too large, this will induce incorrect dynamics at the sector boundaries. Conversely, if *delt* is too small, the simulation will perform few events per sector and spend too much time communicating.

If the *adapt* keyword is *yes*, then *delt* is adjusted as the simulation proceeds. This is done by holding deln fixed, where deln is defined by the equation:

```
deln = delt*pmax
```

and pmax is the maximum propensity per site for one sector. In calculating pmax, we compare all sectors on all processor subdomains, and sites with zero propensity are excluded. The value of deln can be calculated from *delt*, or it can be specified directly using the *deln* keyword. If *deln* is specified as 0 (the default), then deln is computed from the initial state of the system. As with delt, larger values of deln may result in errors at sector boundaries, but small values require more communication. For high accuracy, deln ~ 1 should be adequate.

Restrictions: none

Related commands:

app_style, solve_style

Default:

The option defaults are mask = no, strict = no, kmc = no, adapt = no, delt = 1.0, and deln = 0.0.

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

temperature command

Syntax:

```
temperature T
```

- T = value of temperature used in Boltzmann factor (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

Related commands: none

Default:

The default temperature is 0.0.

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