

# **SPPARKS Users Manual**

Stochastic Parallel PArTicle Kinetic Simulator

<http://www.cs.sandia.gov/~sjplimp/spparks.html> – Sandia National Laboratories

Copyright (2008) Sandia Corporation. This software and manual is distributed under the GNU General Public License.

# Table of Contents

SPPARKS Documentation.....	1
1. Introduction.....	2
1.1 What is SPPARKS.....	2
1.2 SPPARKS features.....	3
Pre- and post-processing:.....	3
1.4 Open source distribution.....	3
1.4 Acknowledgments and citations.....	4
2. Getting Started.....	5
2.1 What's in the SPPARKS distribution.....	5
2.2 Making SPPARKS.....	5
2.3 Making SPPARKS with optional packages.....	7
2.4 Building SPPARKS as a library.....	8
2.5 Running SPPARKS.....	8
2.6 Command-line options.....	9
3. Commands.....	11
3.1 SPPARKS input script.....	11
3.2 Parsing rules.....	12
3.3 Input script structure.....	12
3.4 Commands listed by category.....	13
3.5 Individual commands.....	13
4. How-to discussions.....	15
4.1 Running multiple simulations from one input script.....	15
4.2 Coupling SPPARKS to other codes.....	16
5. Example problems.....	18
6. Performance & scalability.....	19
7. Additional tools.....	20
8. Modifying & extending SPPARKS.....	21
Application styles.....	22
Diagnostic styles.....	23
Input script commands.....	23
Solve styles.....	23
9. Errors.....	25
9.1 Common problems.....	25
9.2 Reporting bugs.....	26
9.3 Error & warning messages.....	26
Errors:.....	26
Warnings:.....	30
add_reaction command.....	31
add_species command.....	32
app_style chemistry command.....	33
app_style diffusion command.....	34
app_style diffusion/table command.....	34
app_style diffusion/nonlinear command.....	34
app_style erbium command.....	36
app_style ising command.....	37
app_style ising/single command.....	37
app_style membrane command.....	39
app_style pore command.....	41

# Table of Contents

app_style pore/nonlinear command.....	41
app_style potts command.....	42
app_style potts/neigh command.....	42
app_style potts/neighonly command.....	42
app_style potts/variable command.....	42
app_style command.....	44
app_style surface command.....	48
app_style test/group command.....	51
clear command.....	53
count command.....	54
diag_style cluster command.....	55
diag_style cluster2d command.....	55
diag_style cluster3d command.....	55
app_style command.....	55
diag_style energy command.....	57
diag_style energy2d command.....	57
diag_style energy3d command.....	57
diag_style eprof3d command.....	58
diag_style erbium command.....	59
diag_style command.....	60
dump command.....	62
echo command.....	64
ecoord command.....	65
event command.....	66
if command.....	68
include command.....	69
inclusion command.....	70
jump command.....	71
label command.....	72
log command.....	73
next command.....	74
print command.....	76
run command.....	77
sector command.....	78
seed command.....	80
shell command.....	81
app_style command.....	83
app_style command.....	84
solve_style command.....	85
app_style command.....	87
stats command.....	88
sweep command.....	90
temperature command.....	92
variable command.....	93
volume command.....	97

# SPPARKS Documentation

(10 Feb 2009 version of SPPARKS)

SPPARKS stands for Stochastic Parallel PARticle Kinetic Simulator.

SPPARKS is a kinetic Monte Carlo (KMC) code designed to run efficiently on parallel computers using both KMC and Metropolis Monte Carlo algorithms. It was developed at Sandia National Laboratories, a US Department of Energy facility, with funding from the DOE. It is an open-source code, distributed freely under the terms of the GNU Public License (GPL).

The developers of SPPARKS are [Steve Plimpton](#), Aidan Thompson, and Alex Slepoy. They can be contacted at [sjplimp@sandia.gov](mailto:sjplimp@sandia.gov), [athomps@sandia.gov](mailto:athomps@sandia.gov), and [alexander.slepoy@nnsa.doe.gov](mailto: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](#)
  - 1.1 [What is SPPARKS](#)
  - 1.2 [SPPARKS features](#)
  - 1.3 [Open source distribution](#)
  - 1.4 [Acknowledgments and citations](#)
2. [Getting started](#)
  - 2.1 [What's in the SPPARKS distribution](#)
  - 2.2 [Making SPPARKS](#)
  - 2.3 [Making SPPARKS with optional packages](#)
  - 2.4 [Building SPPARKS as a library](#)
  - 2.5 [Running SPPARKS](#)
  - 2.6 [Command-line options](#)
3. [Commands](#)
  - 3.1 [SPPARKS input script](#)
  - 3.2 [Parsing rules](#)
  - 3.3 [Input script structure](#)
  - 3.4 [Commands listed by category](#)
  - 3.5 [Commands listed alphabetically](#)
4. [How-to discussions](#)
5. [Example problems](#)
6. [Performance & scalability](#)
7. [Additional tools](#)
8. [Modifying & Extending SPPARKS](#)
9. [Errors](#)
  - 9.1 [Common problems](#)
  - 9.2 [Reporting bugs](#)
  - 9.3 [Error & warning messages](#)
10. [Future plans](#)

# 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](#)

### 1.2 [SPPARKS features](#)

### 1.3 [Open source distribution](#)

### 1.4 [Acknowledgments and citations](#)

---

## 1.1 What is SPPARKS

SPPARKS is a kinetic Monte Carlo (KMC) code that has algorithms for both rejection-free KMC and null-event or rejection KMC (rKMC). In a generic sense its KMC and rKMC solvers catalog a list of "events", each with an associated "probability", choose a single event to perform, and advance time by the correct amount. Events may be chosen individually at random or a sweep of a geometric lattice can be performed to select possible events in a more ordered fashion.

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

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

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

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:

- Diffusion model
- Ising model
- Potts model
- Membrane model
- Biochemical reaction network model

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

- linear,  $O(N)$
- tree,  $O(\log N)$
- 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](http://www.gnu.org) or [www.opensource.org](http://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 application 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](mailto:sjplimp@sandia.gov), [athomps@sandia.gov](mailto:athomps@sandia.gov), [alexander.slepoy@nnsa.doe.gov](mailto: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](#)
  - [2.2 Making SPPARKS](#)
  - [2.3 Making SPPARKS with optional packages](#)
  - [2.4 Building SPPARKS as a library](#)
  - [2.5 Running SPPARKS](#)
  - [2.6 Command-line options](#)
- 

### 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.tflow`, 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.

(3) Building for a Macintosh.

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

---

## **2.3 Making SPPARKS with optional packages**

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](#)
  - [3.2 Parsing rules](#)
  - [3.3 Input script structure](#)
  - [3.4 Commands listed by category](#)
  - [3.5 Commands listed alphabetically](#)
- 

### 3.1 SPPARKS input script

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

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

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

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

does something different than this sequence:

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

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

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

(3) Sometimes command B will use values that can be set by command A. This means command A must precede command B in the input script if it is to have the desired effect.

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

Many input script errors are detected by SPPARKS and an ERROR or WARNING message is printed. [This section](#) gives more information on what errors mean. The documentation for each command lists restrictions on how the command can be used.

---

## 3.2 Parsing rules

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

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

- (1) If the line ends with a `"` character (with no trailing whitespace), the command is assumed to continue on the next line. The next line is concatenated to the previous line by removing the `"` character and newline. This allows long commands to be continued across two or more lines.
  - (2) All characters from the first `#` character onward are treated as comment and discarded.
  - (3) The line is searched repeatedly for `$` characters which indicate variables that are replaced with a text string. If the `$` is followed by curly brackets, then the variable name is the text inside the curly brackets. If no curly brackets follow the `$`, then the variable name is the character immediately following the `$`. Thus `${myTemp}` and `$x` refer to variable names "myTemp" and "x". See the [variable](#) command for details of how strings are assigned to variables and how they are substituted for in input scripts.
  - (4) The line is broken into "words" separated by whitespace (tabs, spaces). Note that words can thus contain letters, digits, underscores, or punctuation characters.
  - (5) The first word is the command name. All successive words in the line are arguments.
  - (6) Text with spaces can be enclosed in double quotes so it will be treated as a single argument. See the [dump](#) [modify](#) or [fix print](#) commands for examples. A `#` or `$` character that in text between double quotes will not be treated as a comment or substituted for as a variable.
- 

## 3.3 Input script structure

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

A SPPARKS input script typically has 3 parts:

- choice of application, solver, sweeper
- settings
- run a simulation

The last 2 parts can be repeated as many times as desired. I.e. run a simulation, change some settings, run some more, etc. Each of the 3 parts is now described in more detail. Remember that almost all the commands need only be used if a non-default value is desired.

### (1) Choice of application, solver, sweep method

Use the [app\\_style](#), [solve\\_style](#), and [sweep](#) commands to setup the kind of simulation you wish to run. Note that sweeping is only relevant to applications that define a geometric lattice of event sites and only if you wish to perform rejection kinetic Monte Carlo updates.

### (2) Settings

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

The [diag\\_style](#) command can also be used to setup various diagnostic computations to perform during a simulation.

### (3) Run a simulation

A kinetic or Metropolis Monte Carlo simulation is performed using the [run](#) command.

---

## 3.4 Commands listed by category

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

Initialization commands:

[app\\_style](#), [solve\\_style](#), [sweep](#), [seed](#)

Application–specific commands:

[add\\_reaction](#), [add\\_species](#), [count](#), [ecoord](#), [event](#), [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.

<a href="#">add_reaction</a>	<a href="#">add_species</a>	<a href="#">app_style</a>	<a href="#">clear</a>	<a href="#">count</a>	<a href="#">diag_style</a>
<a href="#">dump</a>	<a href="#">echo</a>	<a href="#">ecoord</a>	<a href="#">event</a>	<a href="#">if</a>	<a href="#">include</a>



<a href="#">inclusion</a>	<a href="#">jump</a>	<a href="#">label</a>	<a href="#">log</a>	<a href="#">next</a>	<a href="#">print</a>
<a href="#">run</a>	<a href="#">sector</a>	<a href="#">seed</a>	<a href="#">shell</a>	<a href="#">solve_style</a>	<a href="#">stats</a>
<a href="#">sweep</a>	<a href="#">temperature</a>	<a href="#">variable</a>	<a href="#">volume</a>		

---

Application styles. See the [app\\_style](#) command for one–line descriptions of each style or click on the style itself for a full description:

<a href="#">chemistry</a>	<a href="#">diffusion</a>	<a href="#">diffusion/table</a>	<a href="#">diffusion/nonlinear</a>	<a href="#">erbium</a>	<a href="#">ising</a>	<a href="#">membrane</a>	<a href="#">pore</a>
<a href="#">pore/nonlinear</a>	<a href="#">potts</a>	<a href="#">test/group</a>					

---

Solve styles. See the [solve\\_style](#) command for one–line descriptions of each style or click on the style itself for a full description:

<a href="#">group</a>	<a href="#">linear</a>	<a href="#">tree</a>
-----------------------	------------------------	----------------------

---

Diagnostic styles. See the [diag\\_style](#) command for one–line descriptions of each style or click on the style itself for a full description:

<a href="#">cluster</a>	<a href="#">energy</a>	<a href="#">eprof3d</a>	<a href="#">erbium</a>
-------------------------	------------------------	-------------------------	------------------------

## 4. How-to discussions

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

### 4.1 [Running multiple simulations from one input script](#)

### 4.2 [Coupling SPPARKS to other codes](#)

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 visualization 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](#)

As illustrated by the application example, these options are referred to in the SPPARKS documentation as the "style" of a particular command.

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

---

## Application styles

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

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

site_energy	compute energy of a lattice site
site_event_rejection	perform an event with null-bin rejection (for rKMC)
site_propensity	compute propensity of all events on a site (for KMC)
site_event	perform a kinetic Monte Carlo event (for KMC)
input_app	perform application-specific input
init_app	perform application-specific initialization

For an on-lattice application, you also need to define various flags in the constructor of your application, to insure proper operation with the "KMC solvers'\_solve.html and rejection KMC [sweep methods](#). These are the flags, all of which have default values set in app\_lattice.cpp:

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

## Diagnostic styles

Diagnostic classes compute some form of analysis periodically during a simulation. See the [diag\\_style](#) command for details.

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

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

---

## Input script commands

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

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

command	operations performed by the new command
---------	---

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

---

## Solve styles

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

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

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

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

---



---

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

## 9. Errors

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

[9.1 Common problems](#)

[9.2 Reporting bugs](#)

[9.3 Error & warning messages](#)

---

### 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  $\geq$  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.  $A + B \rightarrow C$ .

### 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:

<a href="#">add_reaction</a>	define a chemical reaction
<a href="#">add_species</a>	define a chemical species
<a href="#">count</a>	specify molecular count of a species
<a href="#">stats</a>	output of system info
<a href="#">volume</a>	specify volume of the chemical reactor

**Restrictions:** none

**Related commands:** none

**Default:** none

---

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

## app\_style diffusion command

## app\_style diffusion/table command

## app\_style diffusion/nonlinear command

### Syntax:

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

- style = *diffusion* or *diffusion/table* or *diffusion/nonlinear*
- fraction = fraction of sites to occupy initially
- see the [app\\_style](#) command for additional keywords that can be appended

### Examples:

```
app_style diffusion 0.2 lattice tri 1.0 50 50
app_style diffusion/nonlinear 0.2 lattice tri 1.0 50 50
```

### Description:

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

The Hamiltonian representing the energy of site I for the *diffusion* and *diffusion/table* styles is as follows:

$$H = - \text{Sum}_j \text{delta}_{ij}$$

where  $\text{Sum}_j$  is a sum over all the neighbor sites of site I and  $\text{delta}_{ij}$  is 1 if the spin of sites I and J are the same and 0 otherwise.

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

$$H = \text{Eng}(\text{Sum}_j \text{delta}_{ij})$$

where  $H = 0$  if a site is a vacancy. If a site is occupied,  $\text{Sum}_j$  is the sum over all its neighbor sites and  $\text{delta}_{ij}$  is 1 if site J is occupied and 0 otherwise. The function  $\text{Eng}()$  is a tabulated function specified by the user via the [ecoord](#) command. The inner sum is effectively the coordination number of site I, and the energy of site I is modeled as a function of the coordination number.

The relationship between the 3 variants of `app_style diffusion` is as follows. Styles *diffusion* and *diffusion/table* are the same except for how events and probabilities are stored and accessed internally. Their results should be statistically the same, although not exactly the same, due to differences in how events are chosen by random numbers.

The style *diffusion/nonlinear* should give exactly the same answers as *diffusion/table* if the tabulated function specified by the [ecoord](#) command is set up so that  $E_0 = N$ ,  $E_1 = N-1$ , ...  $E_{N-1} = 1$ ,  $E_N = 0$ .  $N$  = the number of neighbors of each lattice site, i.e. the maximum coordination number. In this scenario, the energy is a linear

function of coordination number, which is effectively the model used by the *diffusion* and *diffusion/table* styles. Note that while the dynamics of the 2 models will be identical, the output energy of the system will be 2x larger in the *diffusion/table* case versus the *diffusion/nonlinear* case. This is because the first Hamiltonian formulas above double-counts the energy of a occupied site neighboring a vacancy site.

Of course, if the specified [ecoord](#) values are not specified as a linear function, then the *diffusion/nonlinear* style will evolve differently.

---

These applications are general lattice application; see the [app\\_style](#) command for further discussion. The lattice must be specified by the appended *lattice* keyword with its associated values, as discussed on the doc page for the [app\\_style](#) command.

This application performs Kawasaki dynamics, meaning the spins on two neighboring sites are swapped. Equivalently, an atom hops from an occupied site to a vacancy site.

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

For solution by a KMC algorithm, the possible events an occupied site can perform are swaps with vacant neighbor sites. The probability of each such event is  $\min[1, \exp(-dE/kT)]$ , where  $dE = E_{\text{final}} - E_{\text{initial}}$ . For the *diffusion* and *diffusion/table* styles,  $E$  is the sum of the energy for the site and its neighbor. For the *diffusion/nonlinear* style,  $E$  is the sum of the energy for the site and its neighbor, and also the energy of the neighbors of each of the 2 sites involved.

For solution by a Metropolis algorithm, the spin is flipped to its opposite state and  $dE = E_{\text{final}} - E_{\text{initial}}$  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.

---

The following additional commands are defined by these applications. The [ecoord](#) command is only defined by the *diffusion/nonlinear* application.

<a href="#">dump</a>	output of lattice snapshots
<a href="#">ecoord</a>	energy as a function of coordination
<a href="#">stats</a>	output of system info
<a href="#">temperature</a>	set Monte Carlo temperature

**Restrictions:** none

**Related commands:**

[app\\_style ising](#), [app\\_style pore](#)

**Default:** none

## app\_style erbium command

### Syntax:

```
app_style erbium frac keyword values ...
```

- `erbium` = application style name
- `frac` = fraction of tetrahedral sites that will initially be hydrogen
- see the [app\\_style](#) command for additional keywords that can be appended

### Examples:

```
app_style erbium 0.1 lattice file tmp.lattice site 2 0
```

### Description:

This application simulates a model on a 3-fold lattice which consists of an fcc lattice with additional tetrahedral and octahedral interstitial sites. The 3-fold lattice should be created using the `erbium_lattice.py` tool in the `tools` directory and read in by the `lattice` keyword of the [app\\_style](#) command.

This application initializes sites on the lattice in the following way. All fcc sites are for Erbium atoms. All octahedral sites are initially vacant. A fraction of the tetrahedral sites, as specified by *frac*, are initialized as Hydrogen atoms; the remainder are vacant.

The [event](#) command is used to define what kinds of diffusive hops occur in the model. These can include correlated hops where a central site coordinates a change of atoms on two of its neighbors. Events that transmute a Hydrogen atom to a Helium are also possible.

---

The following additional commands are defined by this application.

<a href="#">dump</a>	output of lattice snapshots
<a href="#">event</a>	definition of an event on the 3-fold lattice
<a href="#">stats</a>	output of system info
<a href="#">temperature</a>	set Monte Carlo temperature

**Restrictions:** none

**Related commands:**

[diag\\_style erbium](#)

**Default:** none

## app\_style ising command

### app\_style ising/single command

#### Syntax:

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

- style = *ising* or *ising/single*
- see the [app\\_style](#) command for additional keywords that can be appended to the *ising* style

#### Examples:

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

#### Description:

This application evolves 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:

$$H = - \text{Sum}_j \text{delta}_{ij}$$

where  $\text{Sum}_j$  is a sum over all the neighbor sites of site I and  $\text{delta}_{ij}$  is 1 if the spin of sites I and J are the same and 0 otherwise.

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

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

For solution by a KMC algorithm, a site event is a spin flip and its probability is  $\min[1, \exp(-dE/kT)]$ , where  $dE = E_{\text{final}} - E_{\text{initial}}$  using the Hamiltonian defined above for the energy of the site, and T is the temperature of the system defined by the [temperature](#) command (which includes the Boltzmann constant k implicitly).

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

---

The following additional commands are defined by this application:

<a href="#">dump</a>	output of lattice snapshots
<a href="#">stats</a>	output of system info
<a href="#">temperature</a>	set Monte Carlo temperature



**Restrictions:** none

**Related commands:**

[app\\_style](#) [potts](#)

**Default:** none

## app\_style membrane command

### Syntax:

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

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

### Examples:

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

### Description:

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

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

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

$$H = - \mu x_i - \text{Sum}_j (w11 a_{ij} + w01 b_{ij})$$

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

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

For solution by a KMC algorithm, a site event is a spin flip from a lipid to fluid state or vice versa. The probability of the event is  $\min[1, \exp(-dE/kT)]$ , where  $dE = E_{\text{final}} - E_{\text{initial}}$  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  $dE = E_{\text{final}} - E_{\text{initial}}$  is calculated, as is a uniform random number R between 0 and 1. The new state is accepted if  $R < \min[1, \exp(-dE/kT)]$ , else it is rejected.

The following additional commands are defined by these applications:

<a href="#">dump</a>	output of lattice snapshots
<a href="#">inclusion</a>	specify which sites are proteins
<a href="#">stats</a>	output of system info
<a href="#">temperature</a>	set Monte Carlo temperature

**Restrictions:** none

**Related commands:** none

**Default:** none

---

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

## app\_style pore command

## app\_style pore/nonlinear command

### Syntax:

```
app_style style xc yc zc diameter thickness keyword values ...
```

- style = *pore* or *pore/nonlinear*
- xc,yc,zc = coordinates of center point of pore
- diameter = xy diameter of cylindrical pore aligned along z axis
- thickness = z thickness of thin film which the pore spans
- see the [app\\_style](#) command for additional keywords that can be appended

### Examples:

```
app_style pore 10 10 10 4 10 lattice fcc 1.0 20 20 20
```

### Description:

This application is exactly the same as the [app\\_style diffusion](#) application, except for the way the lattice of occupied/vacant sites is initialized.

A single cylindrical pore in a thin-film membrane is setup. The pore is aligned with the z-axis and the thin film is periodic in the x and y dimensions.

The *pore* style is equivalent to the "app\_style diffusion/table" style. The *pore/nonlinear* style is equivalent to the "app\_style diffusion/nonlinear" style.

---

The following additional commands are defined by these applications. The *ecoord* command is only defined by the *pore/nonlinear* application.

<a href="#">dump</a>	output of lattice snapshots
<a href="#">ecoord</a>	energy as a function of coordination
<a href="#">stats</a>	output of system info
<a href="#">temperature</a>	set Monte Carlo temperature

**Restrictions:** none

**Related commands:**

[app\\_style diffusion](#)

**Default:** none

## app\_style potts command

## app\_style potts/neigh command

## app\_style potts/neighonly command

## app\_style potts/variable command

### Syntax:

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

- style = *potts* or *potts/neigh* or *potts/neighonly* or *potts/variable*
- Q = number of spin states
- see the [app\\_style](#) command for additional keywords that can be appended to the *potts* and *potts/variable* styles

### Examples:

```
app_style potts 100 spin lattice sq/4n 1.0 50 50
app_style potts/neigh 20 spin lattice sq/4n 1.0 50 50
app_style potts/variable 100 neigh lattice sq/4n 1.0 50 50 site 1 1
```

### Description:

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

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

$$H = - \sum_j \text{delta}_{ij}$$

where  $\sum_j$  is a sum over all the neighbor sites of site I and  $\text{delta}_{ij}$  is 1 if the spin of sites I and J are the same and 0 otherwise.

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

The *potts/variable* application is identical to the *potts* application except that the *site* keyword must be used to define the per-site quantities stored on the lattice. The *site* keyword is described with the [app\\_style](#) command.

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

For solution by a KMC algorithm, a site event is a spin flip and its probability is  $\min[1, \exp(-dE/kT)]$ , where  $dE = E_{\text{final}} - E_{\text{initial}}$  using the Hamiltonian defined above for the energy of the site, and T is the temperature of the system defined by the [temperature](#) command (which includes the Boltzmann constant k implicitly).

For solution by a rKMC algorithm, the various styles use different rejection-based algorithms. For the *potts* and *potts/variable* styles, a random spin from 1 to Q is chosen. For the *potts/neigh* style, a spin is chosen randomly from the values held by neighbor sites and a null-bin of a size which extends the possible events up to the maximum number of neighbors. For example, imagine a site has 12 neighbors and the 12 sites have 4 different spin values. Then each of the 4 neighbor spin values will be chosen with 1/12 probability and the null bin will be chosen with 8/12 probability. For the *potts/neighborly* style, the null bin is discarded, so in this case each of the 4 spin values will be chosen with 1/4 probability. In all the cases,  $dE = E_{\text{final}} - E_{\text{initial}}$  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.

Need to discuss rules for wild flips for KMC and rKMC.

---

The following additional commands are defined by these applications:

<a href="#">dump</a>	output of lattice snapshots
<a href="#">stats</a>	output of system info
<a href="#">temperature</a>	set Monte Carlo temperature

**Restrictions:** none

**Related commands:**

[app\\_style ising](#)

**Default:** none

## app\_style command

### Syntax:

app\_style style args keyword values keyword values ...

- style = *diffusion* or *ising* or *pore* 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* or *input*

*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 *random/3d*

*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 neighbors 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

*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

*file* param = filename = read lattice and connectivity from file

filename = name of file (see file format below)

*site* values = Nint Ndouble

Nint = # of integer quantities to store per site

Ndouble = # of double quantities to store per site

*input* value = infile

infile = filename for file containing initial state of all lattice sites

### Examples:

```
app_style diffusion ... lattice fcc 1.0 100 100 100
app_style ising ... lattice sq/4n 1.0 100 100
app_style ising ... lattice sq/4n 1.0 100 100 input restart.state
app_style pore ... lattice fcc 1.0 100 100 100
app_style potts ... lattice file tmp.lattice
app_style potts/variable ... lattice random/2d 1000 10.0 10.0 3.0
app_style membrane ... lattice tri 1.0 100 50
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.

Here is the list of on-lattice applications SPPARKS currently includes. See the doc page for each application for details:

- [diffusion](#) = vacancy exchange diffusion model
- [ising](#) = Ising model
- [membrane](#) = membrane model of lipid,water,protein
- [pore](#) = surface diffusion around thin-film pore
- [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](#)

The off-lattice applications in SPPARKS can only be evolved via a kinetic Monte Carlo (KMC) solver, specified by the [solve\\_style](#) command. On-lattice applications can be evolved by either a KMC solver or a rejection kinetic Monte Carlo (rKMC) solver, specified by the [sweep](#) command. Not all on-lattice applications support both choices.

The KMC algorithm is sometimes called rejection-free KMC or the N-fold way or the Gillespie algorithm in the MC literature. The application defines a list of "events" and associated rates for each event. The solver chooses the next event, and the application updates the system accordingly. 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 rKMC algorithm picks successive lattice sites via some method (see the [sweep](#) command), and an event on that site which it accepts or rejects. This is sometimes called null-event MC in the literature. The application again defines the "events" for each site and associated rates which influence the acceptance or rejection.

For both time evolution methods (KMC and rKMC) the rules for how events are defined and are accepted or rejected are discussed in the doc pages for the individual applications.

Here is a table of the different kinds of solvers and options that can be used for on-lattice applications in SPPARKS. Serial and parallel refer to running on one or many processors. Sector vs no-sector is what is set by the [sector](#) command. The rKMC options are set by the [sweep](#) command.

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

Note that the choice of *color* can also be *color/strict* and that masking can also be turned on for rKMC algorithms via the [sweep](#) command if the lo-level application supports it.

---

For on-lattice applications, there are several keyword/value pairs that may be used as part of the `app_style` command.



The *lattice* keyword must be specified for on–lattice applications as it determines the kind and size of lattice used. For example,

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

means use a 2d square lattice of size 100x100 with 4 neighbors per lattice site. A variety of lattice types and neighbor stencils can be 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). 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 on–lattice applications, 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.

---

For on-lattice applications, you can initialize the values stored on each lattice site explicitly by using the *input* keyword. This can be useful for restarting a simulation from the dump file generated by a previous simulation. See the "dump2input.py" script in the tools directory for a way to convert a dump file to the input format described below.

Normally, if the *input* keyword is not used, the application will initialize the lattice in some random manner.

The file specified with the *input* keyword should have the following format. As before, the comments (#) need not be included in the file.

```
comment          # 1st line is skipped
Nsite nvalue      # Nsite = total # of lattice sites, nvalue = values/site
                  # skipped line
1 value1 value2 ...
2 value1 value2 ...
...
Nsites value1 value2 ... # Nsite lines
```

In the body of the file, there is one line per lattice site. These lines can be in any order. The first field on the line is the ID of the site, which should be an integer from 1 to Nsite. There should be "nvalue" remaining fields on each line. These values are assigned to that site. How many values there should be (nvalue) depends on the application. For those with a single integer value (Nint = Ndouble = 0, nvalue = 1), just list that value. If the *site* keyword is used (Nint and/or Ndouble != 0, nvalue = Nint+Ndouble), then list the integer value(s) first, followed by the double value(s).

---

**Restrictions:** none

**Related commands:** none

**Default:**

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

## app\_style surface command

### Syntax:

app\_style surface keyword values ...

- surface = application style name
- zero or more keyword/value pairs may be appended
- keyword = *xcos* or *zcos* or *yfix* or *lattice*

```

xcos values = yfoot yheight xmin xmax
zcos values = yfoot yheight zmin zmax
yfix values = ymax
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 random/3d
  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 neighbors 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
  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
  file param = filename = read lattice and connectivity from file
  filename = name of file (see file format below)

```

- see the [app\\_style](#) command for additional keywords that can be appended

### Examples:

```
app_style surface xcos 51.1 0.0 0.0 40.0 &                                zcos 100.0 5.0 0.0 40.0 &
```

### Description:

This application sets up a y-surface and simulates the evolution of the surface. The keyword *xcos* means that the surface is cosine-shape in the x-direction. The parameters *yfoot*, *yheight*, *xmin*, and *xmax* represents respectively the bottom y coordinate, the height in y direction, and the x-coordinates of the left and the right valleys of the cosine function so that sites above the function is un-occupied. The keyword *zcos* and its parameters have similar meanings except that they are applied in the z-direction. The keyword *yfix* has an argument *ymax*. Any species below is not allowed to move so that only the top surface is simulated.

The energy of the system is expressed as a pairwise summation of the bonding energy  $E_b$  = bondener:

$$E = 0.5 \sum_i \sum_{j, j \neq i} E_b$$

Users should supply the bonding energy in their input file. For example:

```
bondener value
```

Note that the value should be negative in the definition here as formation of bonds reduces the energy.

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, the possible events a occupied site  $i$  can perform are (a) regular jumps where its occupant hops to a nearest neighbor vacant site  $j$ , or (b) Schwoebel jumps where the occupant at  $i$  jumps through a nearest vacant site  $j_1$  to a second nearest vacant site  $k$  around an occupied nearest site  $j_2$ . The probability for an event to occur is proportional to the corresponding jump frequency  $\Gamma$ . This in turn is determined by a energy barrier  $Q = e_{\text{barrier}}$  for regular jumps and  $Q = e_{\text{Schwoebel}}$  for Schwoebel jumps, the atom vibration frequency  $\nu = \text{vibrafreq}$ , and the energy difference before and after the jump,  $dE = E_{\text{final}} - E_{\text{initial}}$ . For  $dE \leq 0$ ,  $\Gamma = \nu \exp(-Q/kT)$ , and for  $dE > 0$ ,  $\Gamma = \nu \exp(-(Q+dE)/kT)$ , where  $k$  is Boltzmann constant and  $T$  is temperature (in unit K). The energy difference can be conveniently calculated as  $dE = E_b * (N_j - N_i)$ , where  $N_j$  and  $N_i$  are total numbers of occupied nearest neighbor sites at site  $j$  and  $i$  respectively. User should supply values of `vibrafreq`, `ebarrier`, and `eSchwoebel` using the following commands in the input file:

```
vibrafreq      value
ebarrier       value Nmin_to
eSchwoebel     value Nmax_from Nmin_to
```

Here `value` are the values for the corresponding parameters. For `ebarrier`, `Nmin_to` is an integer number used to control the jump. When the total number of nearest occupied neighbors at destination is less than `Nmin_to`, the jump is not allowed. For `eSchwoebel`, `Nmax_from` and `Nmin_to` are two integer numbers used to control the jump. When the total number of nearest occupied neighbors at origin is larger than `Nmax_from` or when the total number of nearest occupied neighbors at destination is less than `Nmin_to`, the jump is not allowed. `Nmin_to = 0` means that no constraints is imposed on the jumps, `Nmin_to = 1` would disallow evaporation, `Nmin_to = 3` (2 in 2D) would mean that atom always jump to a stable cradle site, etc. Similarly, large `Nmax_from` value (e.g., `Nmax_from = 11`) would mean that atoms that are not on the surface cannot make Schwoebel jumps, etc. Future work will implement energy barrier tables so that each of the jumps is associated with a separate energy barrier.

For solution by a Metropolis algorithm, the spin is flipped to its opposite state and  $dE = E_{\text{final}} - E_{\text{initial}}$  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.

---

The following additional commands are defined by this application.

<a href="#">dump</a>	output of lattice snapshots
<a href="#">ecoord</a>	energy as a function of coordination
<a href="#">stats</a>	output of system info
<a href="#">temperature</a>	set Monte Carlo temperature

**Restrictions:** none

**Related commands:**

[app\\_style diffusion](#)

**Default:** none

## app\_style test/group command

### Syntax:

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

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

*lomem* value = *yes* or *no*

### Examples:

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

### Description:

This 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 between  $-\text{delta}$  and  $+\text{delta}$ . Since delta is specified as a percentage, this means  $\text{pold} * (1 - \text{delta}/100) \leq \text{pnew} \leq \text{pold} * (1 + \text{delta}/100)$ . Delta must be less than 100.

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

This application can only be evolved using a kinetic Monte Carlo (KMC) algorithm. You must thus define a KMC solver to be used with the application via the [solve\\_style](#) command

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

```
run 10000
```

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

---

The following additional command is defined by this application:

<a href="#">stats</a>	output of system info
-----------------------	-----------------------

**Restrictions:** none

**Related commands:**

[solve\\_style](#) group

**Default:**

The default value is lomem = no.

---

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

## 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 keyword value keyword value ...
```

- style = *cluster* or *cluster2d* or *cluster3d*
- zero or more keyword/value pairs may be appended
- see the [diag\\_style](#) command for additional keywords that can be appended to a diagnostic command and which must appear before these keywords
- keyword = *filename* or *dump*

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

### Examples:

```
diag_style cluster
diag_style cluster2d
diag_style cluster3d
diag_style cluster3d stats no delt 1.0 filename cluster3d.a.0.1.dat dump opendx cluster3d.a.0.1.dump
```

### Description:

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

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

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

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

Clustering Analysis for Lattice (diag\_style cluster) nglobal = nprocs =

----- Time = ncluster = id ivalue dvalue size

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

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

The *dump* keyword causes the cluster ID for each site to be printed out in snapshot format which can be used for visualization purposes. The cluster IDs are arbitrary integers such that two sites have the same ID if and only if they belong to the same cluster. The *standard* setting generates LAMMPS-style. For *cluster2d* and *cluster3d* styles only two values are printed for each site: site index and cluster ID. For the *cluster* style, three additional values are printed: the x, y, and z coordinate of the site (for 2d lattices, z=0). These files can be visualized with various tools in the [LAMMPS package](#) and the [Pizza.py package](#).

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

### Restrictions:

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 keyword value keyword value ...
```

- style = *energy* or *energy2d* or *energy3d*
- see the [diag\\_style](#) command for additional keywords that can be appended to a diagnostic command

### Examples:

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

### 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 keyword value keyword value ...
```

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

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

### Examples:

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

### Description:

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

The *axis* keyword specifies 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 erbium command

### Syntax:

```
diag_style erbium keyword value keyword value ...
```

- erbium = style name of this diagnostic
- zero or more keyword/value pairs may be appended
- see the [diag\\_style](#) command for additional keywords that can be appended to a diagnostic command and which must appear before these keywords
- keyword = *list*

```
list values = er or h or he or vac or events or sN or dN or tN
er,h,he,vac = counts of how many lattice sites of this type currently exist
events = total # of events currently defined for all sites
sN,dN,tN = cumulative # of events for this reaction that have occurred
```

### Examples:

```
diag_style erbium stats yes list h he vac events s1 d1 t2
```

### Description:

The erbium diagnostic prints out statistics about the system being modeled by [app\\_style erbium](#). The values will be printed as part of [stats](#) output. You must use the "stats yes" keyword/value which is discussed with the [diag\\_style](#) command in order to enable this output.

Following the *list* keyword you can list one or more of the listed values, in any order. The *sN*, *dN*, and *tN* values refer to events defined by the [event](#) command. The letter "s" means reactions involving a single site, "d" means double reactions involving 2 sites, and "t" means triple reactions involving 3 sites. The N refers to which reaction (from 1 to the number of that type of reaction). I.e. "t2" means the 2nd 3-site reaction defined in your input script. Note that the statistical values printed for *sN*, *dN*, and *tN* are cumulative counts from the beginning of the simulation run.

### Restrictions:

This command can only be used as part of the [app\\_style erbium](#) application.

### Related commands:

[diag\\_style](#), [stats](#)

**Default:** none

## diag\_style command

### Syntax:

diag\_style style keyword value keyword value ...

- style = *cluster* or *cluster2d* or *cluster3d* or *eprof3d* or *energy* or *energy2d* or *energy3d*
- zero or more keyword/value pairs may be appended

keyword = *delt* or *logfreq* or *stats*

*delt* values = delta

delta = time increment between evaluations of the diagnostic (seconds)

*logfreq* values = N factor

N = number of repetitions per interval

factor = scale factor between interval

*stats* values = *yes* or *no*

yes/no = provide output to stats line

- see doc pages for individual diagnostic commands for additional keywords. These diagnostic-specific keywords must come after all of the standard keywords

### Examples:

```
diag_style cluster stats no delt 1.0
diag_style eprof3d stats no delt 0.01 logfreq 7 10.0
diag_style energy2d
```

### Description:

This command invokes a diagnostic calculation. Currently, diagnostics can only be defined for on-lattice applications. See the [app\\_style](#) command for an overview of such applications. 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 lattices

---

Diagnostics may provide one or more values that are appended to other statistical output and printed to the screen and log file via the [stats](#) command. We call this stats output. In addition, the diagnostic may optionally write more extensive output to files if requested by diagnostic-specific keywords.

The *stats* keyword controls whether or not the diagnostic appends values to the statistical output. If *stats* is set to yes, then the *delt* and *logfreq* keywords may not be used, as the output interval is controlled by the *stats* command. If *stats* is set to no, then the *delt* keyword specifies *Delta*, the interval of time between each diagnostic calculation. Using the *logfreq* keyword will cause the diagnostic to run at varying intervals during the course of a simulation. There will be *N* outputs per interval where the size of each interval scales up by *factor*

each time. *Delta* is the time between outputs in the first (smallest) interval.

For example, this command

```
diag_style eprof3d stats no delt 0.1 logfreq 7 10.0
```

will perform its computation at these times:

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

This command

```
diag_style enegy eprof3d stats no delt 0.1 logfreq 1 2.0
```

will perform its computation at these times:

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

---

**Restrictions:** none

**Related commands:**

[stats](#)

**Default:**

The stats setting is yes.



## dump command

### Syntax:

```
dump delta filename keyword values field1 field2 ...
```

- `delta` = time increment between dumps (seconds) `filename` = name of file to dump snapshots to `keyword` = `logfreq`

`logfreq values = N factor`

`N` = number of repetitions per interval

`factor` = scale factor between interval

- `field` = `id` or `lattice` or `x` or `y` or `z` or `energy` or `propensity` or `iN` or `dN`

### Examples:

```
dump 0.25 tmp.dump
dump 5.0 snap.ising id lattice energy
dump 1.0 snap.erbium logfreq 9 10.0 id i1 i2 x y z
```

### Description:

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

Using the `logfreq` keyword will produce statistical output at varying intervals during the course of a simulation. There will be  $N$  outputs per interval where the size of each interval scales up by *factor* each time. *Delta* is the time between outputs in the first (smallest) interval.

For example, this command

```
dump 0.1 snap.erbium logfreq 7 10.0 id i1 i2 x y z
```

will dump snapshots at these times:

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

This command

```
dump 0.1 snap.erbium logfreq 1 2.0 id i1 i2 x y z
```

will dump snapshots at these times:

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

By default (if no fields are listed), the output values for each site are the "id lattice x y z". This is in the format of a [LAMMPS dump file](#) which can thus be read-in by the [Pizza.py toolkit](#), converted to other formats, or used for visualization.

If fieldss are listed then only those quantities will be printed for each lattice site.

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

The *lattice* value is typically the integer state of the lattice, e.g. the spin value, assuming the application uses the lattice array. Some applications store multiple values per lattice site; see the *site* option of the [app\\_style](#) command. You can specify these values be dumped as *iN* or *dN*, where i = integer, d = double, and N = which value to print (1,2,3,...).

The *x*, *y*, *z* values are the coordinates assigned to the lattice site. The *energy* value is what is computed by the `energy()` function in the application. Likewise for the *propensity* value which can be thought of as the relative probability for that site to perform a KMC event. Note that if you are doing Metropolis MC and not kinetic MC, no propensity is defined.

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

### **Restrictions:**

This command can only be used as part of the lattice-based applications. See the [app\\_style](#) command for further details.

### **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
```

## ecoord command

### Syntax:

```
ecoord N eng
```

- N = coordination number
- eng = energy of site with this coordination number (energy units)

### Examples:

```
ecoord 8 5.6  
ecoord 0 1.0e20
```

### Description:

This command sets the energy of an occupied site in a lattice as a function of coordination number, where coordination = the number of occupied neighbor sites. Typically this command should be used  $N_{\max}+1$  times with N varying from 0 to  $N_{\max}+1$ , when  $N_{\max}$  is the number of neighbor sites for each lattice site.

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

### Restrictions:

This command can only be used as part of the [app\\_style diffusion/nonlinear](#) or [app\\_style pore/nonlinear](#) applications.

### Related commands:

[app\\_style diffusion/nonlinear](#)

**Default:** none

## event command

### Syntax:

```
event Nsite site1 site2 site3 old1 old2 old3 rate new1 new2 new3
```

- Nsite = number of lattice sites involved in the event = 1,2,3
- site1,site2,site3 = *fcc* or *tet* or *oct*
- old1,old2,old3 = *er* or *h* or *he* or *vac*
- rate = rate constant for the event (inverse seconds or energy units)
- new1,new2,new3 = *er* or *h* or *he* or *vac*

### Examples:

```
event      1 tet h 1.78279E-9 he
event      1 oct h 1.78279E-9 he

event      2 tet tet h vac 0.98 vac h
event      2 tet oct h vac 1.89 vac h
event      2 tet oct vac h 0.68 h vac
event      2 tet tet he vac 0.49 vac he
event      2 oct oct he vac 1.49 vac he

event      3 tet oct oct h vac h 0.62 h h vac
event      3 tet oct tet h vac he 1.31 he h vac
event      3 tet oct tet he h vac 0.16 h vac he
event      3 tet oct oct h vac he 0.88 he h vac
event      3 tet oct oct he h vac 0.16 h vac he
```

### Description:

This command defines an event for the "app\_style erbium" application. It can be an event involving one, two, or three lattice sites, as specified by *Nsite*. The first site is the central site which owns the event. The other 2 sites (if specified) are neighbors of the central site.

[App\\_style erbium](#) operates on a 3-fold lattice which contains fcc, tetrahedral, and octahedral sites. The *site1*, *site2*, and *site3* settings specify which kinds of sites are involved in the event: *fcc* or *tet* or *oct*. If *Nsite* = 1, then only *site1* is specified. If *Nsite* = 2, then only *site1* and *site2* are specified.

The *old1*, *old2*, and *old3* settings specify what kinds of atoms must be on those sites in order for the event to possibly take place. The possible atoms are *er* for Erbium, *h* for Hydrogen, *he* for Helium, and *vac* for a vacant site. E.g. in the first example above, a Hydrogen atom must be on a tetrahedral site for the event to be possible.

The *rate* setting determines the relative rate at which the event will occur. For *Nsite*=1 events, the units are inverse seconds. For *Nsite*=2 or *Nsite*=3 events, the units are energy, which is converted into a rate by the application, via the formula:

$$\text{rate} = \exp(-\text{energy}/kT)$$

where T is the [temperature](#) you have specified.

In this case the *rate* setting should be in the energy units defined by the application's Hamiltonian and should be

consistent with the units used in the [temperature](#) command.

The *new1*, *new2*, and *new3* settings specify what the atoms on each of the sites will become if the event takes place. As with the *old* settings, the possible atoms are *er* for Erbium, *h* for Hydrogen, *he* for Helium, and *vac* for a vacant site. E.g. in the first example above, a Hydrogen atom on a tetrahedral site transmutes into a Helium atom if the event takes place.

Note that the set of Nsite=1,2,3 events listed above are a reasonably full description of an Erbium diffusion model.

**Restrictions:** none

This command can only be used as part of the [app\\_style erbium](#) application.

**Related commands:**

[app\\_style erbium](#)

**Default:** none

## if command

### Syntax:

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

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

### Examples:

```
if ${steps} > 1000 then exit
if $x <= $y then "print X is smaller = $x" else "print Y is smaller = $y"
if ${eng} > 0.0 then "timestep 0.005"
if ${eng} > ${eng_previous} then "jump file1" else "jump file2"
```

### Description:

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

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

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

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

**Restrictions:** none

**Related commands:**

[variable](#)

**Default:** none

## include command

### Syntax:

```
include file
```

- file = filename of new input script to switch to

### Examples:

```
include newfile  
include in.run2
```

### Description:

This command opens a new input script file and begins reading SPPARKS commands from that file. When the new file is finished, the original file is returned to. Include files can be nested as deeply as desired. If input script A includes script B, and B includes A, then SPPARKS could run for a long time.

If the filename is a variable (see the [variable](#) command), different processor partitions can run different input scripts.

**Restrictions:** none

### Related commands:

[variable](#), [jump](#)

**Default:** none



## inclusion command

### Syntax:

```
inclusion x y z r
```

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

### Examples:

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

### Description:

This command defines protein sites on a lattice and can only be used by [app\\_style membrane](#) applications.

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

**Restrictions:** none

This command can only be used as part of the [app\\_style pore](#) applications.

### Related commands:

[app\\_style membrane](#)

**Default:** none

## jump command

### Syntax:

```
jump file label
```

- file = filename of new input script to switch to
- label = optional label within file to jump to

### Examples:

```
jump newfile
jump in.run2 runloop
```

### Description:

This command closes the current input script file, opens the file with the specified name, and begins reading SPPARKS commands from that file. The original file is not returned to, although by using multiple jump commands it is possible to chain from file to file or back to the original file.

Optionally, if a 2nd argument is used, it is treated as a label and the new file is scanned (without executing commands) until the label is found, and commands are executed from that point forward. This can be used to loop over a portion of the input script, as in this example. These commands perform 10 runs, each of 10000 steps, and create 10 dump files named file.1, file.2, etc. The [next](#) command is used to exit the loop after 10 iterations. When the "a" variable has been incremented for the tenth time, it will cause the next jump command to be skipped.

```
variable a loop 10
label loop
run 5.0
next a
jump in.1j 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 temperature is now $t"
```

### Description:

Print a text string to the screen and logfile. The text string must be a single argument, so it should be enclosed in double quotes if it is more than one word. If variables are included in the string, they will be evaluated and their current values printed.

If you want the print command to be executed multiple times (with changing variable values) then the print command could appear in a section of the input script that is looped over (see the [jump](#) and [next](#) commands).

See the [variable](#) command for a description of *equal* style variables which are typically the most useful ones to use with the print command. Equal-style variables can calculate formulas involving mathematical operations, or references to other [variables](#).

**Restrictions:** none

**Related commands:**

[variable](#)

**Default:** none

## 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



## sector command

### Syntax:

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

- flag = *yes* or *no*
- zero or more keyword/value pairs may be appended
- keyword = *tstop* or *nstop*

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

### Examples:

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

### Description:

This command partitions the portion of the simulation domain owned by each processor into sectors or sub-domains. It can only be used for [on-lattice applications](#). Typically, it is used in a parallel simulation, to enable parallelism, but it can also be used on a single processor. For 2d lattices, each processor's sub-domain is partitioned into quadrants. In 3d it is partitioned into octants.

If sectors are turned on via a *yes* setting, then a kinetic Monte Carlo (KMC) or rejection KMC (rKMC) algorithm is performed in the following manner. Events or sites are selected within the first sector on each processor, via a [solver](#) or [sweeping method](#). Communication is then done between processors to update sector boundaries. Then all processors 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 processed one at a time, followed by the appropriate communication. This procedure insures events occurring on one processor do not conflict with events performed by other processors.

The optional keywords for this command determine how much time is spent on each sector (i.e. how many events are performed) before moving to the next sector. Note that using sectors turns an exact KMC or rKMC algorithm into an approximate one, in the spirit of [Amar](#). This is because events are occurring within a sector while the state of the system on the boundary of the sector is held frozen. If the time-per-sector is too large, this will require less communication but will induce incorrect dynamics at the sector boundaries. Conversely, if the time-per-sector is too small, the simulation will perform few events per sector and spend too much time communicating.

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

If the *nstop* keyword is used, it sets the average number of events (or attempts) per site. For example, an *nstop* value of 2.0 means attempt 2 events per site for an rKMC algorithm. For a KMC algorithm, this is converted into a time by computing the maximum propensity of all sites within any sector in the simulation domain. In the KMC case, this means that if the total propensity of the system decreases as the simulation proceeds (e.g. grain growth

occurs), then the effective time per sweep will increase in an adaptive way. Said another way, the number of events per sweep will remain roughly constant, as the time per event increases. In the rKMC case, the time per attempt is constant due to the use of a null-bin, so there is no adaptivity.

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

Note that it makes no sense to define *tstop* and *nstop* together since they defined the time-per-sector in different ways. If both are used, the last setting takes precedence.

**Restrictions:**

This command can only be used as part of on-lattice applications as specified by the [app\\_style](#) command.

**Related commands:**

[app\\_style](#), [solve\\_style](#), [sweep](#)

**Default:**

The option defaults are  $nstop = 1.0$ .

---

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

## seed command

### Syntax:

```
seed Nvalue
```

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

### Examples:

```
seed 5838959
```

### Description:

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

**Restrictions:** none

**Related commands:** none

**Default:** none

## shell command

### Syntax:

```
shell style args
```

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

```
cd arg = dir
    dir = directory to change to
mkdir args = dir1 dir2 ...
    dir1,dir2 = one or more directories to create
mv args = old new
    old = old filename
    new = new filename
rm args = file1 file2 ...
    file1,file2 = one or more filenames to delete
rmdir args = dir1 dir2 ...
    dir1,dir2 = one or more directories to delete
```

### Examples:

```
shell cd sub1
shell cd ..
shell mkdir tmp1 tmp2 tmp3
shell rmdir tmp1
shell mv log.lammps hold/log.1
shell rm TMP/file1 TMP/file2
```

### Description:

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

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

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

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

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

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

### Restrictions:

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

**Related commands:** none

**Default:** none

## app\_style command

### Syntax:

```
app_style style args
```

- style = application style name
- args = args

### Examples:

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

### Description:

This command ...

**Restrictions:** none

### Related commands:

[variable](#), ...

**Default:** none

## app\_style command

### Syntax:

```
app_style style args
```

- style = application style name
- args = args

### Examples:

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

### Description:

This command ...

**Restrictions:** none

### Related commands:

[variable](#), ...

**Default:** none

## solve\_style command

### Syntax:

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

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

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

### Examples:

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

### Description:

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

A KMC solver picks events for your application to perform from a list of events and their associated probabilities. It does this using the standard [Gillespie](#) or [BKL](#) algorithm which also computes a timestep during which the chosen event occurs. 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  $O(\log N)$ .

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  $\geq hi$ .

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 = *logfreq*

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

### Examples:

```
stats 0.1
stats 1.0 logfreq 7 10.0
```

### Description:

Print statistics to the screen and log file every so many seconds during a simulation. The quantities printed are elapsed CPU time followed by those provided by the [application](#), followed by those provided by any [diagnostics](#) you have defined.

Typically the application reports only the number of events or sweeps executed, followed by the simulation time, but other application-specific quantities may also be reported. Quantities such as the total energy of the system can be included in the output by creating diagnostics via the [diag\\_style](#) command.

Using the *logfreq* keyword will produce statistical output at varying intervals during the course of a simulation. There will be *N* outputs per interval where the size of each interval scales up by *factor* each time. *Delta* is the time between outputs in the first (smallest) interval.

For example, this command

```
stats 0.1 logfreq 7 10.0
```

will produce output at these times:

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

This command

```
stats 0.1 logfreq 1 2.0
```

will produce output at these times:

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

---

### Restrictions:

See the doc pages for quantities provided by particular [app\\_style](#) and [diag\\_style](#) commands for further details.

**Related commands:**

[dump](#), [diag](#)

**Default:** none

## sweep command

### Syntax:

```
sweep style keyword value ...
```

- style = *random* or *raster* or *color* or *color/strict*
- zero or more keyword/value pairs may be appended
- keyword = *mask*

```
mask value = yes or no  
yes/no = mask out sites than cannot change
```

### Examples:

```
sweep random  
sweep raster mask yes ...
```

### Description:

Use a rejection kinetic Monte Carlo (rKMC) algorithm for an [on-lattice application](#). If rKMC is not used then a kinetic Monte Carlo (KMC) algorithm must be used as defined by the [solve\\_style](#) command.

The rKMC algorithm in SPPARKS selects sites on a lattice in an order determined by this command and requests that the application perform events. The application defines the geometry and connectivity of the lattice, what the possible events are, and defines their rates and acceptance/rejection criteria.

The ordering of selected sites is also affected by the [sector](#) command, which partitions each processor's portion of the simulation domain into sectors which are quadrants (2d) or octants (3d). In this case, the ordering described below is within each sector. Sectors are looped over one at a time, interleaved by communication of lattice values inbetween.

For the *random* style, sites are chosen randomly, one at a time.

For the *raster* style, a sweep of the lattice is done, as a loop over all sites in a pre-determined order, e.g. a triple loop over i,j,k for a 3d cubic lattice.

For the *color* style, lattice sites are partitioned into sub-groups or colors which are non-interacting in the sense that events on two sites of the same color can be performed simultaneously without conflict. This enables parallelism since events on all sites of the same color can be attempted simultaneously. Similar to sectors, the colors are looped over, interleaved by communication of lattice values inbetween.

The *color/strict* style is the same as the *color* style except that random numbers are generated in a way that is independent of the processor which generates them. Thus SPPARKS should produce the same answer, independent of how many processors are used. This can be useful in debugging an application.

If the application supports it, the *mask* keyword can be set to *yes* to skip sites which cannot perform an event due to the current value of the site and its neighbors. Enabling masking should not change the answer given by a simulation (in a statistical sense); it only offers a computational speed-up. For example, sites in the interior of grains in a Potts grain-growth model may have no potential of flipping their value. Masking can only be set to *yes* if the [temperature](#) is set to 0.0, since otherwise there is a finite probability of any site performing an event.

**Restrictions:**

This command can only be used as part of on-lattice applications as specified by the [app\\_style](#) command.

Not all lattice styles and applications support the *color* and *color/strict* styles. Not all applications support the *mask* option.

**Related commands:**

[app\\_style](#), [solve\\_style](#), [sector](#)

**Default:**

The option defaults are mask = no.

## temperature command

### Syntax:

```
temperature T
```

- T = value of temperature for the Monte Carlo simulation (energy units)

### Examples:

```
temperature 2.0
```

### Description:

This command sets the temperature as used in various applications. The typical usage would be as part of a Boltzmann factor that alters the propabilities of event acceptance and rejection.

The units of the specfied temperature should be consistent with how the application defines energy. E.g. if used in a Boltzmann factor where a  $kT$  factor scales the energy of a Hamiltonian defined by the application, then this command is really defining  $kT$  and the specified value should have the units of energy as computed by the Hamiltonian.

**Restrictions:** none

This command can only be used as part of applications that allow for a temperature to be specified. See the doc pages for individual applications defined by the [app\\_style](#) command for further details.

**Related commands:** none

### Default:

The default temperature is 0.0.

## 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