SPPARKS Users Manual

27 Nov 2024 version

https://spparks.github.io - Sandia National Laboratories

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

SPPARKS Documentation

27 Nov 2024 version

Version info:

The SPPARKS "version" is the date when it was released, such as 12 Jun 2018. SPPARKS is updated continuously. Whenever we fix a bug or add a feature, we release it immediately, and post a notice on this page of the WWW site. Each dated copy of SPPARKS contains all the features and bug-fixes up to and including that version date. The version date is printed to the screen and logfile every time you run SPPARKS. It is also in the file src/version.h and in the SPPARKS directory name created when you unpack a tarball.

- If you browse the HTML or PDF doc pages on the SPPARKS WWW site, they always describe the most current version of SPPARKS.
- If you browse the HTML or PDF doc pages included in your tarball, they describe the version you have.

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), or sometimes by request under the terms of the GNU Lesser General Public License (LGPL).

The SPPARKS website has more information about the code and publications that desribe it. The current SPPARKS developers are John Mitchell (Sandia National Labs) and Steve Plimpton. They can be contacted at jamitch@sandia.gov and sjplimp@gmail.com respectively. Past developers and other significant code contributores are listed on the Authors page of the website.

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

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

PDF file of the entire manual, generated by htmldoc

1. Introduction

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

1.1 What is SPPARKS

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

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

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

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

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

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

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

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

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

1.2 SPPARKS features

These are three kinds of applications in SPPARKS:

- on-lattice
- off-lattice
- general

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

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

- diffusion model
- Ising model
- Potts model in many variants
- membrane model
- sintering model

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

- Metropolis atomic relaxation model

The set of general applications currently in SPPARKS are:

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

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

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

Pre- and post-processing:

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

1.3 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), or sometimes by request under the terms of the GNU Lesser General Public License (LGPL). This is often referred to as open-source distribution - see www.gnu.org or www.opensource.org for more details. The legal text of the GPL or LGPL is in the LICENSE file that is included in the SPPARKS distribution.

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

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

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

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

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

In the spirit of an open-source code, if you use SPPARKS for something useful or if you fix a bug or add a new feature or applicaton to the code, let us know. We would like to include your contribution in the released version of the code and/or advertise your success on our WWW page.

1.4 Acknowledgments and citations

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

The Authors page of the SPPARKS website lists the developers and their contact info, along with others who have contributed code and expertise to the developement of SPPARKS.

2. Getting Started

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

2.1 What's in the SPPARKS distribution

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

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

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

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

2.2 Making SPPARKS

This section has the following sub-sections:

Read this first:

Building SPPARKS can be non-trivial. You will likely need to edit a makefile, there are compiler options, additional libraries can be used (MPI, JPEG), 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, say *serial* or *mpi* or *linux*, then type one of the commands:

```
make serial
make mpi
gmake linux
```

Try the "serial" and "mpi" targets first, since they are generic and should typically work on any machine, assuming you have the GNU g++ compiler (for the serial version) and MPI installed (for the mpi version).

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_serial or spk_mpi is produced, you're done; it's your lucky day.

IMPORTANT NOTE: You need a C++ compiler that is C++11 compliant to build SPPARKS. Almost all current C++ compilers are; you just need to use a -std=c++11 flag when compiling, as in the src/MAKE/Makefile.machine files provided with SPPARKS.

Common 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.

(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) The "compiler/linker settings" section lists compiler and linker settings for your C++ compiler, including path and optimization flags. You can use g++, the open-source GNU compiler, which is available on all Unix systems. You can also use mpicc which will typically be available if MPI is installed on your system, though you should check which actual compiler it wraps. You can also point to a specific compiler; for example see MAKE/Makefile.spencer.gnu where an environment variable MPI_HOME is used to specify path to mpicxx and mpicc compilers.

Vendor compilers often produce faster code. On boxes with Intel CPUs, we suggest using the commercial Intel icc compiler, which can be downloaded from Intel's compiler site.

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

The DEPFLAGS setting is what triggers the C++ compiler to create a dependency list for a source file. This speeds re-compilation when source (*.cpp) or header (*.h) files are edited. Some compilers do not support dependency file creation, or may use a different switch than -D. GNU g++ works with -D. If your compiler can't create dependency files (a long list of errors involving *.d files), then you'll need to create a Makefile.foo patterned after Makefile.storm, which uses different rules that do not involve dependency files.

(3) The "system-specific settings" section has 3 parts.

(3.a) The SPK_INC variable is used to include options that turn on system-dependent ifdefs within the SPPARKS code. The settings that are currently recogized are:

- -DSPPARKS_GZIP
- -DSPPARKS_JPEG
- -DSPPARKS_SMALLBIG
- -DSPPARKS_BIGBIG
- -DSPPARKS_SMALLSMALL

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

If you use -DSPPARKS_JPEG, the dump image command will be able to write out JPEG image files. If not, it will only be able to write out text-based PPM image files. For JPEG files, you must also link SPPARKS with a JPEG library. See section (3.c) below for more details on this.

Use at most one of the -DSPPARKS_SMALLBIG, -DSPPARKS_BIGBIG, -DSPPARKS_SMALLSMALL settings. The default is -DSPPARKS_SMALLBIG. These settings refer to use of 4-byte (small) vs 8-byte (big) integers within SPPARKS, as specified in src/spktype.h. The only reason to use the BIGBIG setting is to enable simulation of systems with more than 2 billion sites. Normally, the only reason to use SMALLSMALL is if your machine does not support 64-bit integers. See the Additional build tips section below for more details on these settings.

(3.b) The 3 MPI variables are used to specify an MPI library to build SPPARKS with.

If you want SPPARKS to run in parallel, you must have an MPI library installed on your platform. If you use an MPI-wrapped compiler, such as "mpicc" to build SPPARKS, you can probably leave these 3 variables blank. If you do not use "mpicc" as your compiler/linker, then you need to specify where the mpi.h file (MPI_INC) and the MPI library (MPI_PATH) is found and its name (MPI_LIB).

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

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 a true MPI library installed on your system. See the Makefile.serial file for how to specify the 3 MPI variables. 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 this 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.

(3.c) The 3 JPG variables are used to specify a JPEG library which SPPARKS uses when writing a JPEG file via the dump image command. These can be left blank if you are not using the -DSPPARKS_JPEG switch discussed above in section (3.a).

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

That's it. Once you have a correct Makefile.foo and you have pre-built any other libraries it will use (e.g. MPI, JPEG), all you need to do from the src directory is type one of these 2 commands:

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) Changing the SPPARKS size limits via -DSPPARKS_SMALLBIG or -DSPPARKS_BIGBIG or -DSPPARKS_SMALLSMALL

As explained above, any of these 3 settings can be specified on the SPK_INC line in your low-level src/MAKE/Makefile.foo.

The default is -DSPPARKS_SMALLBIG which allows for systems with up to 2^31 sites (about 2 billion). This is because the site IDs are stored in 32-bit integers.

To allow for larger systems, compile with -DSPPARKS_BIGBIG. This stores site IDs in 64-bit integers. This enables systems with up to 2^63 sites (about 9e18).

If your system does not support 8-byte integers, you will need to compile with the -DSPPARKS_SMALLSMALL setting. This will restrict the total number of sites to 2^31 (about 2 billion), as well as store some simulation statistics in 4-byte integers.

Note that in src/lmptype.h there are definitions of all these data types as well as the MPI data types associated with them. The MPI types need to be consistent with the associated C data types, or else SPPARKS will generate a run-time error. As far as we know, the settings defined in src/spktype.h are portable and work on every current system.

In all cases, the size of problem that can be run on a per-processor basis is limited by 4-byte integer storage to 2^31 sites per processor (about 2 billion). This should not normally be a limitation since such a problem would have a huge per-processor memory and would run very slowly in terms of CPU secs per Monte Carlo interation.

Building for a Mac:

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

Building for Windows:

SPPARKS is just C++ with MPI calls, so it should be possible to build it for a Windows box, either using a Linux installation such as cygwin (see src/MAKE/Makefile.cygwin), or importing the source files into Visual Studio C++ and building it there. For the latter you are on your own. The SPPARKS developers do not use Windows. But if you figure out how to do it, or create a Visual Studio project that works, please let us know, and we can release the instructions/files for how to do this as part of SPPARKS.

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".

Currently there is only one optional package: STITCH. It is dicussed more below.

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.

2.3.1 STITCH package

The STITCH package allows SPPARKS to use the Stitch library for I/O, which is included in the SPPARKS distribution in lib/stitch. At some point the Stitch library will have its own website and will also be downloadable

there.

Stitch is an efficient I/O API and database format with a native python interface. *Stitch* files can read in to start a simulation and/or output during a simulation. A novel aspect of *stitch* is that it enables out-of-core computations by building a simulation domain analogously to the way an additive manufactured (AM) part is built. It merges outputs written over time to efficiently construct a much larger simulation domain that would otherwise be impossible to model in one simulation. *Stitching* workflows can be created to perform multiple SPPARKS simulations representing an additive manufacturing process; such simulations can produce huge numbers of lattice sites representing an entire AM build that would otherwise be impossible to simulate due to length scale and computational resource limitations. *Stitch* is intended and primarily focused on microstructural evolution simulations such as welding and additive manufacturing but other applications may be possible.

Building SPPARKS with the STITCH package enables these commands to use stitch-related options:

- dump stitch
- set stitch
- reset_time

See the am_path and stitch sub-directories in the examples directory for models and scripts which use the Stitch library.

You can build SPPARKS with *stitch* support in one of 3 ways.

(1) From the src directory using make

```
% cd spparks/src
% make lib-stitch args="-b"   # build the Stitch library and set links to it
% make yes-stitch             # install the STITCH package
% make mpi                    # or whichever machine target you wish
```

(2) From the lib directory using Install.py

```
% cd spparks/lib
% python Install.py -b        # build the Stitch library and set links to it
% cd spparks/src
% make yes-stitch             # install the STITCH package
% make mpi                    # or whichever machine target you wish
```

(3) Manual build of the Stitch library you have downloaded to your system

```
% cd $STITCHDIR                # STITCHDIR = the Stitch library directory
% make                         # build Stitch with default Makefile
% make -f Makefile.custom      # build Stitch with custom Makefile
% cd spparks/lib/stitch
% ln -s $STITCHDIR liblink     # set two links in SPPARKS lib/stitch
% ln -s $STITCHDIR includelink
% cd spparks/src
% make yes-stitch             # install the STITCH package
% make mpi                    # or whichever machine target you wish
```

To un-install the STITCH package from SPPARKS, do the following:

```
% cd spparks/src
% make no-stitch     # un-install the STITCH package files
% make mpi           # re-build SPPARKS w/out the STITCH package
```

2.4 Building SPPARKS as a library

SPPARKS can be built as either a static or shared library, which can then be called from another application or a scripting language. See this section for more info on coupling SPPARKS to other codes. See this section for more info on wrapping and running SPPARKS from Python.

Static library:

To build SPPARKS as a static library (*.a file on Linux), type

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

where foo is the machine name. This kind of library is typically used to statically link a driver application to SPPARKS, so that you can insure all dependencies are satisfied at compile time. Note that inclusion or exclusion of any desired optional packages should be done before typing "make makelib". The first "make" command will create a current Makefile.lib with all the file names in your src dir. The second "make" command will use it to build SPPARKS as a static library, using the ARCHIVE and ARFLAGS settings in src/MAKE/Makefile.foo. The build will create the file libspparks_foo.a which another application can link to.

Shared library:

To build SPPARKS as a shared library (*.so file on Linux), which can be dynamically loaded, e.g. from Python, type

```
make makeshlib
make -f Makefile.shlib foo
```

where foo is the machine name. This kind of library is required when wrapping SPPARKS with Python; see Section_python for details. Again, note that inclusion or exclusion of any desired optional packages should be done before typing "make makelib". The first "make" command will create a current Makefile.shlib with all the file names in your src dir. The second "make" command will use it to build SPPARKS as a shared library, using the SHFLAGS and SHLIBFLAGS settings in src/MAKE/Makefile.foo. The build will create the file libspparks_foo.so which another application can link to dyamically. It will also create a soft link libspparks.so, which the Python wrapper uses by default.

Note that for a shared library to be usable by a calling program, all the auxiliary libraries it depends on must also exist as shared libraries. This will be the case for libraries included with SPPARKS, such as the dummy MPI library in src/STUBS since they are always built as shared libraries with the -fPIC switch. However, if a library like MPI does not exist as a shared library, the second make command will generate an error. This means you will need to install a shared library version of the package. The build instructions for the library should tell you how to do this.

As an example, here is how to build and install the MPICH library, a popular open-source version of MPI, distributed by Argonne National Labs, as a shared library in the default /usr/local/lib location:

```
./configure --enable-shared
make
make install
```

You may need to use "sudo make install" in place of the last line if you do not have write privileges for /usr/local/lib. The end result should be the file /usr/local/lib/libmpich.so.

Additional requirement for using a shared library:

The operating system finds shared libraries to load at run-time using the environment variable LD_LIBRARY_PATH. So you may wish to copy the file src/libspparks.so or src/libspparks_g++.so (for example) to a place the system can find it by default, such as /usr/local/lib, or you may wish to add the SPPARKS src directory to LD_LIBRARY_PATH, so that the current version of the shared library is always available to programs that use it.

For the csh or tcsh shells, you would add something like this to your ~/.cshrc file:

```
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/home/sjplimp/spparks/src
```

Calling the SPPARKS library:

Either flavor of library (static or shared0 allows one or more SPPARKS objects to be instantiated from the calling program.

When used from a C++ program, all of SPPARKS is wrapped in a SPPARKS_NS namespace; you can safely use any of its classes and methods from within the calling code, as needed.

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

See the sample codes in examples/COUPLE/simple for examples of C++ and C and Fortran codes that invoke SPPARKS thru its library interface. There are other examples as well in the COUPLE directory which are discussed in Section_howto 2 of the manual. See Section_python of the manual for a description of the Python wrapper provided with SPPARKS that operates through the SPPARKS library interface.

The files src/library.cpp and library.h define the C-style API for using SPPARKS as a library. See Section_howto 3 of the manual for a description of the interface and how to extend it for your needs.

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 a section below. 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.

2.7 SPPARKS screen output

As SPPARKS reads an input script, it prints information to both the screen and a log file about significant actions it takes to setup a simulation. When the simulation is ready to begin, SPPARKS performs various initializations and prints the amount of memory (in MBytes per processor) that the simulation requires. An example output is shown here, for the examples/in.potts script run on 4 processors.

```
SPPARKS (11 Dec 2015)
Created box = (0 0 0) to (20 20 20)
  1 by 2 by 2 processor grid
Creating sites ...
  8000 sites
  8000 sites have 26 neighbors
Setting site values ...
  8000 settings made for site
Setting up run ...
Memory usage per processor = 4.375 Mbytes
```

During the run itself, statistical information is printed periodically, for every delta of simulation time, as specified by the stats commmand. When the run concludes, SPPARKS prints final statistical info and a total run time for the simulation.

```
      Time    Naccept    Nreject    Nsweeps         CPU     Energy
         0          0          0          0           0     205912
     10.01      88437    7919563       1001       0.195      72506
        20      94828   15905172       2000       0.379      57038
        30      98345   23901655       3000       0.565      49948
        40     101449   31898551       4000       0.749      44316
     50.01     103978   39904022       5001       0.933      39334
     60.01     105578   47902422       6001        1.12      36902
     70.01     106938   55901062       7001         1.3      34428
        80     108491   63891509       8000        1.49      31668
        90     110211   71889789       9000        1.67      27994
```

```
       100     112074   79887926      10000       1.86      21894
Loop time of 1.86084 on 4 procs
```

It then appends statistics about the breakdown of CPU time for the simulation.

```
Solve time (%) = 1.52001 (81.6842)
Update time (%) = 0 (0)
Comm  time (%) = 0.245275 (13.1809)
Outpt time (%) = 0.0892967 (4.79874)
App   time (%) = 0 (0)
Other time (%) = 0.00625533 (0.336157)
```

3. Commands

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

3.1 SPPARKS input script

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

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

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

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

does something different than this sequence:

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

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

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

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

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

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

3.2 Parsing rules

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

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

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

(2) All characters from the first "#" character onward are treated as comment and discarded.

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

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

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

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

3.3 Input script structure

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

A SPPARKS input script typically has 3 parts:

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

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

(1) Choice of application, solver, sweep method

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

(2) Settings

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

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

(3) Run a simulation

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

3.4 Commands listed by category

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

Initialization commands:

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

Setting commands:

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

Application-specific commands:

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

Output commands:

diag_style, dump, dump image, dump_modify, dump_one, stats, undump

Actions:

run,

Miscellaneous:

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

---

3.5 Individual commands

This section lists all SPPARKS commands alphabetically, with a separate listing below of styles within certain commands. The previous section lists the same commands, grouped by category. Note that commands which are only usable with certain applications are listed in the next section.

app_style	boundary	clear	create_box	create_sites	diag_style
dimension	dump	dump image	dump_modify	dump_one	echo

if	include	jump	label	lattice	log
next	pair_coeff	pair_style	print	processors	read_sites
region	reset_time	run	sector	seed	set
shell	solve_style	stats	sweep	undump	variable

Application-specific commands. These are commands defined only for use by one or more applications. See the command doc page for details. See the various app_style commands in the next section for a listing of all the commands defined for individual applications.

add_reaction	add_species	am_build	am cartesian_layer	am pass	am path
am path_layer	am pathgen	barrier	count	deep_length	deep_width
deposition	diffusion/multiphase	ecoord	elliopsoid_depth	event	inclusion
pin	pulse	temperature	volume	weld_shape_ellipse	weld_shape_teardrop

---

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

am/ellipsoid	chemistry	diffusion	diffusion/multiphase	erbium	ising	ising/single	membrane
phasefield/potts	potts	potts/am/bezier	potts/am/path/gen	potts/am/weld	potts/grad	potts/neigh	potts/neighonly
potts/pin	potts/quaternion	potts/strain	potts/strain/pin	potts/weld	potts/weld/jom	relax	sinter
sos	test/group						

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

group	linear	tree

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

lj/cut

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

array	cluster	diffusion	energy	erbium	propensity
sinter_avg_neck_area	sinter_density	sinter_free_energy_pore	sinter_pore_curvature		

4. How-to discussions

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

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

4.1 Running multiple simulations from one input script

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

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

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

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

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

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

would run 2 independent simulations, one after the other.

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

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

```
jump in.runs
```

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

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

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

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

4.2 Coupling SPPARKS to other codes

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

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

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

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

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

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

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

Examples of driver codes that call SPPARKS as a library are included in the examples/COUPLE directory of the SPPARKS distribution; see examples/COUPLE/README for more details:

- simple: simple driver programs in C++ and C which invoke SPPARKS as a library (NOTE: not yet available)
- lammps_spparks: coupling of SPPARKS and LAMMPS, to couple a kinetic Monte Carlo model for grain growth using MD to calculate strain induced across grain boundaries

This section of the documentation describes how to build SPPARKS as a library. Once this is done, you can interface with SPPARKS either via C++, C, Fortran, or Python (or any other language that supports a vanilla C-like interface). For example, from C++ you could create one (or more) "instances" of 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. See Section_python of the manual for a description of the Python wrapper provided with SPPARKS that operates through the SPPARKS library interface.

The files src/library.cpp and library.h contain the C-style interface to SPPARKS. See Section_howto 3 of the manual for a description of the interface and how to extend it for your needs.

Note that the spparks_open() function that creates an instance of SPPARKS takes an MPI communicator as an argument. This means that instance of SPPARKS will run on the set of processors in the communicator. Thus 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. Or it might instantiate multiple instances of SPPARKS to perform different calculations.

4.3 Library interface to SPPARKS

As described in Section_start 4, SPPARKS can be built as a library, so that it can be called by another code, used in a coupled manner with other codes, or driven through a Python interface.

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

Library.cpp contains these 4 functions:

```
void spparks_open(int, char **, MPI_Comm, void **);
void spparks_close(void *);
void spparks_file(void *, char *);
char *spparks_command(void *, 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 in stand-alone mode 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 is used in subsequent library calls. 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. Or it might instantiate multiple instances of SPPARKS to perform different calculations.

The spparks_close() function is used to shut down an instance of 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 script or single command in an input script. Thus the calling code can read or generate a series of SPPARKS commands one line at a time and pass it thru the library interface to setup a problem and then run it, interleaving the spparks_command() calls with other calls to extract information from SPPARKS, perform its own operations, or call another code's library.

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

```
void *spparks_extract(void *, char *)
double *spparks_energy()
```

These can extract various global or per-site quantities from SPPARKS so that a driver application can access the values or even reset them. See the library.cpp file and its associated header file library.h for details.

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

5. Example problems

The SPPARKS distribution includes an examples sub-directory with several sample problems. Each problem is in a sub-directory of its own. Most are small models that can be run quickly, requiring at most a couple of minutes to run on a desktop machine. Each problem has an input script (in.*) and produces a log file (log.*) and dump file (dump.*) when it runs. A few sample log file outputs on different machines and different numbers of processors are included in the directories to compare your answers to. E.g. a log file like log.potts.foo.P means it ran on P processors of machine "foo".

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

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

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

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

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

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

If you add dump image line(s) to the input script a series of JPG images will be produced by the run. These can be viewed individually or turned into a movie or animated by tools like ImageMagick or QuickTime or various Windows-based tools. See the dump image doc page for more details. E.g. this Imagemagick command would create a GIF file suitable for viewing in a browser.

```
% convert -loop 1 *.jpg foo.gif
```

There is also a COUPLE directory with examples of how to use SPPARKS as a library, either by itself or in tandem with another code or library. See the COUPLE/README file to get started.

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.

Addtional scripts below are distributed with spparks under the tools directory.

- potts_quaternion/cpp_quaternion.py: enables reading spparks quaternion header files
- potts_quaternion/plot_cubic_symmetry_histograms.py: verification plots for disorientation distribution of randomly oriented cubic structures
- potts_quaternion/plot_hcp_symmetry_histograms.py: verification plots for disorientation distribution of randomly oriented hcp structures

8. Modifying & extending SPPARKS

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

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

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

Most of the new features described in this section require you to write a new C++ derived class. Creating a new class requires 2 files, a source code file (*.cpp) and a header file (*.h). The derived class must provide certain methods to work as a new option. Depending on how different your new feature is compared to existing features, you can either derive from the base class itself, or from a derived class that already exists. Enabling SPPARKS to invoke the new class is as simple as adding two lines to the style_user.h file, in the same syntax as other SPPARKS classes are specified in the style.h file.

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

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

```
app_style foo 0.1 3.5
```

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

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

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

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

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

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

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

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

Application styles

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

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

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

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

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

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

Diagnostic styles

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

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

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

Input script commands

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

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

command	operations performed by the new command

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

Solve styles

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

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

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

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

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

9. Errors

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

9.1 Common problems

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

SPPARKS tries to flag errors and print informative error messages so you can fix the problem. Of course SPPARKS cannot figure out your physics mistakes, like choosing too big a timestep or setting up an invalid lattice. If you find errors that SPPARKS doesn't catch that you think it should flag, please send an email to the developers.

If you get an error message about an invalid command in your input script, you can determine what command is causing the problem by looking in the log.spparks file or using the echo command to see it on the screen. For example you can run your script as

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

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

Generally, SPPARKS will print a message to the screen and exit gracefully when it encounters a fatal error. Sometimes it will print a WARNING and continue on; you can decide if the WARNING is important or not. If SPPARKS crashes or hangs without spitting out an error message first then it could be a bug (see this section) or one of the following cases:

SPPARKS runs in the available memory each processor can allocate. All large memory allocations in the code are done via C-style malloc's which will generate an error message if you run out of memory. Smaller chunks of memory are allocated via C++ "new" statements. If you are unlucky you could run out of memory when one of these small requests is made, in which case the code will crash, since SPPARKS doesn't trap on those errors.

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

In parallel, one way SPPARKS can hang is due to how different MPI implementations handle buffering of messages. If the code hangs without an error message, it may be that you need to specify an MPI setting or two (usually via an environment variable) to enable buffering or boost the sizes of messages that can be buffered.

9.2 Reporting bugs

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

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

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

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

9.3 Error & warning messages

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

Errors:

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

Adding site to illegal bin
: Internal SPPARKS error.

All pair coeffs are not set
: Self-explanatory.

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

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

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

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

App did not set dt_sweep
: Internal SPPARKS error.

App does not permit user_update yes
: UNDOCUMENTED

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

App relax requires a pair potential
: Self-explanatory.

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

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

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

App_style specific command before app_style set
: Self-explanatory.

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

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

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

BAD DONE
: UNDOCUMENTED

BAD STENCIL
: UNDOCUMENTED

BIN MISMATCH
: UNDOCUMENTED

Bad neighbor site ID
: UNDOCUMENTED

Bigint setting in spktype.h is invalid
: UNDOCUMENTED

Boundary command after simulation box is defined
: UNDOCUMENTED

Boundary command currently only supported by on-lattice apps
: UNDOCUMENTED

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

COUNT MISMATCH
: UNDOCUMENTED

Can only read Neighbors for on-lattice applications
: UNDOCUMENTED

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

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

Cannot color without a lattice definition of sites
: UNDOCUMENTED

Cannot color without contiguous site IDs
: UNDOCUMENTED

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

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

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

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

Cannot create/grow a vector/array of pointers for %s
: UNDOCUMENTED

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

Cannot dump JPG file
: UNDOCUMENTED

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

Cannot open diag_style cluster dump file
: Self-explanatory.

Cannot open diag_style cluster output file
: Self-explanatory.

Cannot open dump file
: Self-explanatory.

Cannot open file %s
: Self-explanatory.

Cannot open gzipped file
: Self-explantory.

Cannot open input script %s
: Self-explanatory.

Cannot open log.spparks
: Self-explanatory.

Cannot open logfile
: Self-explanatory.

Cannot open logfile %s
: Self-explanatory.

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

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

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

Cannot perform deposition in parallel
: UNDOCUMENTED

Cannot perform deposition with multiple sectors
: UNDOCUMENTED

Cannot read Neighbors after sites already exist
: UNDOCUMENTED

Cannot read Neighbors unless max neighbors is set
: UNDOCUMENTED

Cannot read Sites after sites already exist
: UNDOCUMENTED

Cannot read Values before sites exist or are read
: UNDOCUMENTED

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

Cannot run 1d simulation with nonperiodic Y or Z dimension
: UNDOCUMENTED

Cannot run 2d simulation with nonperiodic Z dimension
: UNDOCUMENTED

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

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

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

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

Cannot use coloring without domain nx,ny,nz defined

UNDOCUMENTED

Cannot use create_sites basis with random lattice

Self-explanatory.

Cannot use diag_style cluster without a lattice defined

This diagnostic uses the lattice style to dump OpenDx files.

Cannot use dump_one for first snapshot in dump file

Self-explanatory.

Cannot use random rejection KMC in parallel with no sectors

Self-explanatory.

Cannot use raster rejection KMC in parallel with no sectors

Self-explanatory.

Cannot use region INF or EDGE when box does not exist

Can only define a region with these parameters after a simulation box has been defined.

Choice of sector stop led to no rKMC events

Self-explanatory.

Color stencil is incommensurate with lattice size

Since coloring induces a pattern of colors, this pattern must fit an integer number of times into a periodic lattice.

Could not find dump ID in dump_modify command

Self-explanatory.

Could not find dump ID in dump_one command

Self-explanatory.

Could not find dump ID in undump command

Self-explanatory.

Create_box command before app_style set

Self-explanatory.

Create_box region ID does not exist

Self-explanatory.

Create_box region must be of type inside

Self-explanatory.

Create_sites command before app_style set

Self-explanatory.

Create_sites command before simulation box is defined

Self-explanatory.

Create_sites region ID does not exist

Self-explanatory.

Creating a quantity application does not support

The application defines what variables it supports. You cannot set a variable with the create_sites command for a variable that isn't supported.

Data file dimension does not match existing box

UNDOCUMENTED

Data file maxneigh setting does not match existing sites

UNDOCUMENTED

Data file number of sites does not match existing sites

UNDOCUMENTED

Data file simluation box different that current box

UNDOCUMENTED

Diag cluster does not work if ncluster > 2^31

UNDOCUMENTED

Diag cluster dvalue in neighboring clusters do not match

Internal SPPARKS error.

Diag cluster ivalue in neighboring clusters do not match

Internal SPPARKS error.

Diag propensity requires KMC solve be performed

Only KMC solvers compute a propensity for sites and the system.

Diag style cluster dump file name too long

Self-explanatory.

Diag style incompatible with app style

The lattice styles of the diagnostic and the on-lattice application must match.

Diag_style cluster incompatible with lattice style

UNDOCUMENTED

Diag_style cluster nx,ny,nz = 0

UNDOCUMENTED

Diag_style command before app_style set

Self-explanatory.

Diag_style diffusion requires app_style diffusion

Self-explanatory.

Diag_style erbium requires app_style erbium

UNDOCUMENTED

Did not assign all sites correctly

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

Did not create correct number of sites

One or more created sites were not assigned to a processor correctly.

Did not reach event propensity threshhold

UNDOCUMENTED

Dimension command after lattice is defined

Self-explanatory.

Dimension command after simulation box is defined

Self-explanatory.

Divide by 0 in variable formula

Self-explanatory.

Dump command before app_style set

Self-explanatory.

Dump command can only be used for spatial applications

Self-explanatory.

Dump image boundary requires lattice app

UNDOCUMENTED

Dump image crange must be set

UNDOCUMENTED

Dump image drange must be set

UNDOCUMENTED

Dump image persp option is not yet supported

UNDOCUMENTED

Dump image requires one snapshot per file

UNDOCUMENTED

Dump image with quantity application does not support

UNDOCUMENTED

Dump requires propensity but no KMC solve performed

Only KMC solvers compute propensity for sites.

Dump_modify command before app_style set

Self-explanatory.

Dump_modify region ID does not exist

UNDOCUMENTED

Dump_modify scolor requires integer attribute for dump image color

UNDOCUMENTED

Dump_modify sdiam requires integer attribute for dump image diameter

UNDOCUMENTED

Dump_one command before app_style set

Self-explanatory.

Dumping a quantity application does not support

The application defines what variables it supports. You cannot output a variable in a dump that isn't supported.

Failed to allocate %ld bytes for array %s

Your SPPARKS simulation has run out of memory. You need to run a smaller simulation or on more processors.

Failed to reallocate %ld bytes for array %s

Your SPPARKS simulation has run out of memory. You need to run a smaller simulation or on more processors.

GHOST IN OWNED BIN

UNDOCUMENTED

Ghost connection was not found

Internal SPPARKS error. Should not occur.

Ghost site was not found

Internal SPPARKS error. Should not occur.

Illegal ... command

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

Incorrect args for pair coefficients

Self-explanatory.

Incorrect lattice neighbor count

Internal SPPARKS error.

Incorrect site format in data file

Self-explanatory.

Incorrect value format in data file

Self-explanatory.

Input line too long after variable substitution

This is a hard (very large) limit defined in the input.cpp file.

Input line too long: %s

This is a hard (very large) limit defined in the input.cpp file.

Invalid attribute in dump text command

UNDOCUMENTED

Invalid color in dump_modify command

UNDOCUMENTED

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 diag_style command

UNDOCUMENTED

Invalid dump image filename

UNDOCUMENTED

Invalid dump image persp value

UNDOCUMENTED

Invalid dump image theta value

UNDOCUMENTED

Invalid dump image zoom value

UNDOCUMENTED

Invalid dump style
: UNDOCUMENTED

Invalid dump_modify threshold operator
: Self-explanatory.

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

Invalid image color range
: UNDOCUMENTED

Invalid image up vector
: UNDOCUMENTED

Invalid keyword in dump command
: Self-explanatory.

Invalid keyword in variable formula
: UNDOCUMENTED

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

Invalid number of sectors
: Self-explanatory.

Invalid pair style
: Self-explanatory.

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

Invalid probability bounds for solve_style group
: Self-explanatory.

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

Invalid region style
: Self-explanatory.

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

Invalid syntax in variable formula
: Self-explanatory.

Invalid value setting in diag_style erbium
: UNDOCUMENTED

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

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

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

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

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

LINK MISMATCH
: UNDOCUMENTED

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

Lattice command before app_style set
: Self-explanatory.

Lattice style does not match dimension
: Self-explanatory.

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

MPI_SPK_BIGINT and bigint in spktype.h are not compatible
: UNDOCUMENTED

MPI_SPK_TAGINT and tagint in spktype.h are not compatible
: UNDOCUMENTED

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

Mismatch in counting for dbufclust
: Self-explanatory.

Must read Sites before Neighbors
: Self-explanatory.

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

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

No Neighbors defined in site file
: UNDOCUMENTED

No Sites defined in site file
: UNDOCUMENTED

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

No solver class defined
: Self-explanatory.

Off-lattice application data file cannot have maxneigh setting
: UNDOCUMENTED

One or more Hamiltonian params are unset
: UNDOCUMENTED

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

PBC remap of site failed
: Internal SPPARKS error.

Pair_coeff command before app_style set
: Self-explanatory.

Pair_coeff command before pair_style is defined
: Self-explanatory.

Pair_style command before app_style set
: Self-explanatory.

Per-processor solve tree is too big
: UNDOCUMENTED

Per-processor system is too big
: UNDOCUMENTED

Periodic box is not a multiple of lattice spacing
: UNDOCUMENTED

Power by 0 in variable formula
: Self-explanatory.

Processor partitions are inconsistent

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

Processors command after simulation box is defined
Self-explanatory.

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

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

Reaction cannot have more than MAX_PRODUCT products
Self-explanatory.

Reaction has no numeric rate
Self-explanatory.

Reaction must have 0,1,2 reactants
Self-explanatory.

Read_sites command before app_style set
Self-explanatory.

Region ID for dump text does not exist
UNDOCUMENTED

Region command before app_style set
Self-explanatory.

Region intersect region ID does not exist
Self-explanatory.

Region union region ID does not exist
Self-explanatory.

Rejection events are not implemented in app
Self-explanatory.

Reset_time command before app_style set
Self-explanatory.

Reuse of dump ID
UNDOCUMENTED

Reuse of region ID
Self-explanatory.

Run command before app_style set
Self-explanatory.

Run upto value is before current time
Self-explanatory.

SITE MISMATCH
UNDOCUMENTED

SITES NOT IN BINS
UNDOCUMENTED

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

Set command before sites exist
Self-explanatory.

Set command region ID does not exist
Self-explanatory.

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

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

Site file has no Sites, Neighbors, or Values
: UNDOCUMENTED

Site not in my bin domain
: Internal SPPARKS error.

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

Smallint setting in spktype.h is invalid
: UNDOCUMENTED

Solve_style command before app_style set
: Self-explanatory.

Species ID %s already exists
: Self-explanatory.

Species ID %s does not exist
: Self-explanatory.

Sqrt of negative in variable formula
: Self-explanatory.

Stats command before app_style set
: Self-explanatory.

Substitution for illegal variable
: Self-explanatory.

System in site file is too big
: UNDOCUMENTED

Tagint setting in spktype.h is invalid
: UNDOCUMENTED

Temperature cannot be 0.0 for app erbium
: UNDOCUMENTED

Threshold for a quantity application does not support
: The application defines what variables it supports. You cannot do a threshold test with the dump command on a variable that isn't supported.

Too many neighbors per site
: Internal SPPARKS error.

Unbalanced quotes in input line
: No matching end double quote was found following a leading double quote.

Undump command before app_style set
: Self-explanatory.

Unexpected end of data file
: Self-explanatory.

Universe/uloop variable count < # of partitions
: A universe or uloop style variable must specify a number of values >= to the number of processor partitions.

Unknown command: %s
: The command is not known to SPPARKS. Check the input script.

Unknown identifier in data file: %s
: Self-explanatory.

Unknown species in reaction command
: Self-explanatory.

Unrecognized command
: The command is assumed to be application specific, but is not known to SPPARKS. Check the input script.

Use of region with undefined lattice
: The lattice command must be used before defining a geometric region.

Variable for dump image center is invalid style

UNDOCUMENTED

Variable for dump image persp is invalid style

UNDOCUMENTED

Variable for dump image phi is invalid style

UNDOCUMENTED

Variable for dump image theta is invalid style

UNDOCUMENTED

Variable for dump image zoom is invalid style

UNDOCUMENTED

Variable name for dump image center does not exist

UNDOCUMENTED

Variable name for dump image persp does not exist

UNDOCUMENTED

Variable name for dump image phi does not exist

UNDOCUMENTED

Variable name for dump image theta does not exist

UNDOCUMENTED

Variable name for dump image zoom does not exist

UNDOCUMENTED

Variable name must be alphanumeric or underscore characters

Self-explanatory.

World variable count doesn't match # of partitions

A world-style variable must specify a number of values equal to the number of processor partitions.

Warnings:

%d propensities were reset to hi value, max hi = %g

UNDOCUMENTED

%d propensities were reset to lo value, max lo = %g

UNDOCUMENTED

Using dump image boundary with spheres

UNDOCUMENTED

SPPARKS Documentation

27 Nov 2024 version

Version info:

The SPPARKS "version" is the date when it was released, such as 12 Jun 2018. SPPARKS is updated continuously. Whenever we fix a bug or add a feature, we release it immediately, and post a notice on this page of the WWW site. Each dated copy of SPPARKS contains all the features and bug-fixes up to and including that version date. The version date is printed to the screen and logfile every time you run SPPARKS. It is also in the file src/version.h and in the SPPARKS directory name created when you unpack a tarball.

- If you browse the HTML or PDF doc pages on the SPPARKS WWW site, they always describe the most current version of SPPARKS.
- If you browse the HTML or PDF doc pages included in your tarball, they describe the version you have.

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), or sometimes by request under the terms of the GNU Lesser General Public License (LGPL).

The SPPARKS website has more information about the code and publications that desribe it. The current SPPARKS developers are John Mitchell (Sandia National Labs) and Steve Plimpton. They can be contacted at jamitch@sandia.gov and sjplimp@gmail.com respectively. Past developers and other significant code contributores are listed on the Authors page of the website.

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

3. Commands

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

3.1 SPPARKS input script

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

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

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

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

does something different than this sequence:

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

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

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

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

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

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

3.2 Parsing rules

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

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

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

(2) All characters from the first "#" character onward are treated as comment and discarded.

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

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

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

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

3.3 Input script structure

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

A SPPARKS input script typically has 3 parts:

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

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

(1) Choice of application, solver, sweep method

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

(2) Settings

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

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

(3) Run a simulation

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

3.4 Commands listed by category

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

Initialization commands:

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

Setting commands:

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

Application-specific commands:

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

Output commands:

diag_style, dump, dump image, dump_modify, dump_one, stats, undump

Actions:

run,

Miscellaneous:

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

---

3.5 Individual commands

This section lists all SPPARKS commands alphabetically, with a separate listing below of styles within certain commands. The previous section lists the same commands, grouped by category. Note that commands which are only usable with certain applications are listed in the next section.

app_style	boundary	clear	create_box	create_sites	diag_style
dimension	dump	dump image	dump_modify	dump_one	echo

if	include	jump	label	lattice	log
next	pair_coeff	pair_style	print	processors	read_sites
region	reset_time	run	sector	seed	set
shell	solve_style	stats	sweep	undump	variable

Application-specific commands. These are commands defined only for use by one or more applications. See the command doc page for details. See the various app_style commands in the next section for a listing of all the commands defined for individual applications.

add_reaction	add_species	am_build	am cartesian_layer	am pass	am path
am path_layer	am pathgen	barrier	count	deep_length	deep_width
deposition	diffusion/multiphase	ecoord	elliopsoid_depth	event	inclusion
pin	pulse	temperature	volume	weld_shape_ellipse	weld_shape_teardrop

---

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

am/ellipsoid	chemistry	diffusion	diffusion/multiphase	erbium	ising	ising/single	membrane
phasefield/potts	potts	potts/am/bezier	potts/am/path/gen	potts/am/weld	potts/grad	potts/neigh	potts/neighonly
potts/pin	potts/quaternion	potts/strain	potts/strain/pin	potts/weld	potts/weld/jom	relax	sinter
sos	test/group						

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

group	linear	tree

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

lj/cut

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

array	cluster	diffusion	energy	erbium	propensity
sinter_avg_neck_area	sinter_density	sinter_free_energy_pore	sinter_pore_curvature		

9. Errors

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

9.1 Common problems

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

SPPARKS tries to flag errors and print informative error messages so you can fix the problem. Of course SPPARKS cannot figure out your physics mistakes, like choosing too big a timestep or setting up an invalid lattice. If you find errors that SPPARKS doesn't catch that you think it should flag, please send an email to the developers.

If you get an error message about an invalid command in your input script, you can determine what command is causing the problem by looking in the log.spparks file or using the echo command to see it on the screen. For example you can run your script as

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

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

Generally, SPPARKS will print a message to the screen and exit gracefully when it encounters a fatal error. Sometimes it will print a WARNING and continue on; you can decide if the WARNING is important or not. If SPPARKS crashes or hangs without spitting out an error message first then it could be a bug (see this section) or one of the following cases:

SPPARKS runs in the available memory each processor can allocate. All large memory allocations in the code are done via C-style malloc's which will generate an error message if you run out of memory. Smaller chunks of memory are allocated via C++ "new" statements. If you are unlucky you could run out of memory when one of these small requests is made, in which case the code will crash, since SPPARKS doesn't trap on those errors.

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

In parallel, one way SPPARKS can hang is due to how different MPI implementations handle buffering of messages. If the code hangs without an error message, it may be that you need to specify an MPI setting or two (usually via an environment variable) to enable buffering or boost the sizes of messages that can be buffered.

9.2 Reporting bugs

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

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

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

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

9.3 Error & warning messages

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

Errors:

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

Adding site to illegal bin
: Internal SPPARKS error.

All pair coeffs are not set
: Self-explanatory.

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

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

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

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

App did not set dt_sweep
: Internal SPPARKS error.

App does not permit user_update yes
: UNDOCUMENTED

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

App relax requires a pair potential
: Self-explanatory.

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

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

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

App_style specific command before app_style set
: Self-explanatory.

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

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

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

BAD DONE
: UNDOCUMENTED

BAD STENCIL
: UNDOCUMENTED

BIN MISMATCH
: UNDOCUMENTED

Bad neighbor site ID
: UNDOCUMENTED

Bigint setting in spktype.h is invalid
: UNDOCUMENTED

Boundary command after simulation box is defined
: UNDOCUMENTED

Boundary command currently only supported by on-lattice apps
: UNDOCUMENTED

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

COUNT MISMATCH
: UNDOCUMENTED

Can only read Neighbors for on-lattice applications
: UNDOCUMENTED

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

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

Cannot color without a lattice definition of sites
: UNDOCUMENTED

Cannot color without contiguous site IDs
: UNDOCUMENTED

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

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

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

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

Cannot create/grow a vector/array of pointers for %s
: UNDOCUMENTED

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

Cannot dump JPG file
: UNDOCUMENTED

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

Cannot open diag_style cluster dump file
: Self-explanatory.

Cannot open diag_style cluster output file
: Self-explanatory.

Cannot open dump file
: Self-explanatory.

Cannot open file %s
: Self-explanatory.

Cannot open gzipped file
: Self-explantory.

Cannot open input script %s
: Self-explanatory.

Cannot open log.spparks
: Self-explanatory.

Cannot open logfile
: Self-explanatory.

Cannot open logfile %s
: Self-explanatory.

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

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

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

Cannot perform deposition in parallel
: UNDOCUMENTED

Cannot perform deposition with multiple sectors
: UNDOCUMENTED

Cannot read Neighbors after sites already exist
: UNDOCUMENTED

Cannot read Neighbors unless max neighbors is set
: UNDOCUMENTED

Cannot read Sites after sites already exist
: UNDOCUMENTED

Cannot read Values before sites exist or are read
: UNDOCUMENTED

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

Cannot run 1d simulation with nonperiodic Y or Z dimension
: UNDOCUMENTED

Cannot run 2d simulation with nonperiodic Z dimension
: UNDOCUMENTED

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

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

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

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

Cannot use coloring without domain nx,ny,nz defined

UNDOCUMENTED

Cannot use create_sites basis with random lattice

Self-explanatory.

Cannot use diag_style cluster without a lattice defined

This diagnostic uses the lattice style to dump OpenDx files.

Cannot use dump_one for first snapshot in dump file

Self-explanatory.

Cannot use random rejection KMC in parallel with no sectors

Self-explanatory.

Cannot use raster rejection KMC in parallel with no sectors

Self-explanatory.

Cannot use region INF or EDGE when box does not exist

Can only define a region with these parameters after a simulation box has been defined.

Choice of sector stop led to no rKMC events

Self-explanatory.

Color stencil is incommensurate with lattice size

Since coloring induces a pattern of colors, this pattern must fit an integer number of times into a periodic lattice.

Could not find dump ID in dump_modify command

Self-explanatory.

Could not find dump ID in dump_one command

Self-explanatory.

Could not find dump ID in undump command

Self-explanatory.

Create_box command before app_style set

Self-explanatory.

Create_box region ID does not exist

Self-explanatory.

Create_box region must be of type inside

Self-explanatory.

Create_sites command before app_style set

Self-explanatory.

Create_sites command before simulation box is defined

Self-explanatory.

Create_sites region ID does not exist

Self-explanatory.

Creating a quantity application does not support

The application defines what variables it supports. You cannot set a variable with the create_sites command for a variable that isn't supported.

Data file dimension does not match existing box

UNDOCUMENTED

Data file maxneigh setting does not match existing sites

UNDOCUMENTED

Data file number of sites does not match existing sites

UNDOCUMENTED

Data file simluation box different that current box

UNDOCUMENTED

Diag cluster does not work if ncluster > 2^31

UNDOCUMENTED

Diag cluster dvalue in neighboring clusters do not match

Internal SPPARKS error.

Diag cluster ivalue in neighboring clusters do not match

Internal SPPARKS error.

Diag propensity requires KMC solve be performed

Only KMC solvers compute a propensity for sites and the system.

Diag style cluster dump file name too long

Self-explanatory.

Diag style incompatible with app style

The lattice styles of the diagnostic and the on-lattice application must match.

Diag_style cluster incompatible with lattice style

UNDOCUMENTED

Diag_style cluster nx,ny,nz = 0

UNDOCUMENTED

Diag_style command before app_style set

Self-explanatory.

Diag_style diffusion requires app_style diffusion

Self-explanatory.

Diag_style erbium requires app_style erbium

UNDOCUMENTED

Did not assign all sites correctly

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

Did not create correct number of sites

One or more created sites were not assigned to a processor correctly.

Did not reach event propensity threshhold

UNDOCUMENTED

Dimension command after lattice is defined

Self-explanatory.

Dimension command after simulation box is defined

Self-explanatory.

Divide by 0 in variable formula

Self-explanatory.

Dump command before app_style set

Self-explanatory.

Dump command can only be used for spatial applications

Self-explanatory.

Dump image boundary requires lattice app

UNDOCUMENTED

Dump image crange must be set

UNDOCUMENTED

Dump image drange must be set

UNDOCUMENTED

Dump image persp option is not yet supported

UNDOCUMENTED

Dump image requires one snapshot per file

UNDOCUMENTED

Dump image with quantity application does not support

UNDOCUMENTED

Dump requires propensity but no KMC solve performed

Only KMC solvers compute propensity for sites.

Dump_modify command before app_style set

Self-explanatory.

Dump_modify region ID does not exist

UNDOCUMENTED

Dump_modify scolor requires integer attribute for dump image color

UNDOCUMENTED

Dump_modify sdiam requires integer attribute for dump image diameter

UNDOCUMENTED

Dump_one command before app_style set

Self-explanatory.

Dumping a quantity application does not support

The application defines what variables it supports. You cannot output a variable in a dump that isn't supported.

Failed to allocate %ld bytes for array %s

Your SPPARKS simulation has run out of memory. You need to run a smaller simulation or on more processors.

Failed to reallocate %ld bytes for array %s

Your SPPARKS simulation has run out of memory. You need to run a smaller simulation or on more processors.

GHOST IN OWNED BIN

UNDOCUMENTED

Ghost connection was not found

Internal SPPARKS error. Should not occur.

Ghost site was not found

Internal SPPARKS error. Should not occur.

Illegal ... command

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

Incorrect args for pair coefficients

Self-explanatory.

Incorrect lattice neighbor count

Internal SPPARKS error.

Incorrect site format in data file

Self-explanatory.

Incorrect value format in data file

Self-explanatory.

Input line too long after variable substitution

This is a hard (very large) limit defined in the input.cpp file.

Input line too long: %s

This is a hard (very large) limit defined in the input.cpp file.

Invalid attribute in dump text command

UNDOCUMENTED

Invalid color in dump_modify command

UNDOCUMENTED

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 diag_style command

UNDOCUMENTED

Invalid dump image filename

UNDOCUMENTED

Invalid dump image persp value

UNDOCUMENTED

Invalid dump image theta value

UNDOCUMENTED

Invalid dump image zoom value

UNDOCUMENTED

Invalid dump style
: UNDOCUMENTED

Invalid dump_modify threshold operator
: Self-explanatory.

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

Invalid image color range
: UNDOCUMENTED

Invalid image up vector
: UNDOCUMENTED

Invalid keyword in dump command
: Self-explanatory.

Invalid keyword in variable formula
: UNDOCUMENTED

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

Invalid number of sectors
: Self-explanatory.

Invalid pair style
: Self-explanatory.

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

Invalid probability bounds for solve_style group
: Self-explanatory.

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

Invalid region style
: Self-explanatory.

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

Invalid syntax in variable formula
: Self-explanatory.

Invalid value setting in diag_style erbium
: UNDOCUMENTED

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

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

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

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

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

LINK MISMATCH
: UNDOCUMENTED

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

Lattice command before app_style set
: Self-explanatory.

Lattice style does not match dimension
: Self-explanatory.

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

MPI_SPK_BIGINT and bigint in spktype.h are not compatible
: UNDOCUMENTED

MPI_SPK_TAGINT and tagint in spktype.h are not compatible
: UNDOCUMENTED

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

Mismatch in counting for dbufclust
: Self-explanatory.

Must read Sites before Neighbors
: Self-explanatory.

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

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

No Neighbors defined in site file
: UNDOCUMENTED

No Sites defined in site file
: UNDOCUMENTED

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

No solver class defined
: Self-explanatory.

Off-lattice application data file cannot have maxneigh setting
: UNDOCUMENTED

One or more Hamiltonian params are unset
: UNDOCUMENTED

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

PBC remap of site failed
: Internal SPPARKS error.

Pair_coeff command before app_style set
: Self-explanatory.

Pair_coeff command before pair_style is defined
: Self-explanatory.

Pair_style command before app_style set
: Self-explanatory.

Per-processor solve tree is too big
: UNDOCUMENTED

Per-processor system is too big
: UNDOCUMENTED

Periodic box is not a multiple of lattice spacing
: UNDOCUMENTED

Power by 0 in variable formula
: Self-explanatory.

Processor partitions are inconsistent

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

Processors command after simulation box is defined
Self-explanatory.

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

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

Reaction cannot have more than MAX_PRODUCT products
Self-explanatory.

Reaction has no numeric rate
Self-explanatory.

Reaction must have 0,1,2 reactants
Self-explanatory.

Read_sites command before app_style set
Self-explanatory.

Region ID for dump text does not exist
UNDOCUMENTED

Region command before app_style set
Self-explanatory.

Region intersect region ID does not exist
Self-explanatory.

Region union region ID does not exist
Self-explanatory.

Rejection events are not implemented in app
Self-explanatory.

Reset_time command before app_style set
Self-explanatory.

Reuse of dump ID
UNDOCUMENTED

Reuse of region ID
Self-explanatory.

Run command before app_style set
Self-explanatory.

Run upto value is before current time
Self-explanatory.

SITE MISMATCH
UNDOCUMENTED

SITES NOT IN BINS
UNDOCUMENTED

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

Set command before sites exist
Self-explanatory.

Set command region ID does not exist
Self-explanatory.

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

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

Site file has no Sites, Neighbors, or Values
: UNDOCUMENTED

Site not in my bin domain
: Internal SPPARKS error.

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

Smallint setting in spktype.h is invalid
: UNDOCUMENTED

Solve_style command before app_style set
: Self-explanatory.

Species ID %s already exists
: Self-explanatory.

Species ID %s does not exist
: Self-explanatory.

Sqrt of negative in variable formula
: Self-explanatory.

Stats command before app_style set
: Self-explanatory.

Substitution for illegal variable
: Self-explanatory.

System in site file is too big
: UNDOCUMENTED

Tagint setting in spktype.h is invalid
: UNDOCUMENTED

Temperature cannot be 0.0 for app erbium
: UNDOCUMENTED

Threshold for a quantity application does not support
: The application defines what variables it supports. You cannot do a threshold test with the dump command on a variable that isn't supported.

Too many neighbors per site
: Internal SPPARKS error.

Unbalanced quotes in input line
: No matching end double quote was found following a leading double quote.

Undump command before app_style set
: Self-explanatory.

Unexpected end of data file
: Self-explanatory.

Universe/uloop variable count < # of partitions
: A universe or uloop style variable must specify a number of values >= to the number of processor partitions.

Unknown command: %s
: The command is not known to SPPARKS. Check the input script.

Unknown identifier in data file: %s
: Self-explanatory.

Unknown species in reaction command
: Self-explanatory.

Unrecognized command
: The command is assumed to be application specific, but is not known to SPPARKS. Check the input script.

Use of region with undefined lattice
: The lattice command must be used before defining a geometric region.

Variable for dump image center is invalid style

UNDOCUMENTED

Variable for dump image persp is invalid style

UNDOCUMENTED

Variable for dump image phi is invalid style

UNDOCUMENTED

Variable for dump image theta is invalid style

UNDOCUMENTED

Variable for dump image zoom is invalid style

UNDOCUMENTED

Variable name for dump image center does not exist

UNDOCUMENTED

Variable name for dump image persp does not exist

UNDOCUMENTED

Variable name for dump image phi does not exist

UNDOCUMENTED

Variable name for dump image theta does not exist

UNDOCUMENTED

Variable name for dump image zoom does not exist

UNDOCUMENTED

Variable name must be alphanumeric or underscore characters

Self-explanatory.

World variable count doesn't match # of partitions

A world-style variable must specify a number of values equal to the number of processor partitions.

Warnings:

%d propensities were reset to hi value, max hi = %g

UNDOCUMENTED

%d propensities were reset to lo value, max lo = %g

UNDOCUMENTED

Using dump image boundary with spheres

UNDOCUMENTED

5. Example problems

The SPPARKS distribution includes an examples sub-directory with several sample problems. Each problem is in a sub-directory of its own. Most are small models that can be run quickly, requiring at most a couple of minutes to run on a desktop machine. Each problem has an input script (in.*) and produces a log file (log.*) and dump file (dump.*) when it runs. A few sample log file outputs on different machines and different numbers of processors are included in the directories to compare your answers to. E.g. a log file like log.potts.foo.P means it ran on P processors of machine "foo".

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

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

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

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

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

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

If you add dump image line(s) to the input script a series of JPG images will be produced by the run. These can be viewed individually or turned into a movie or animated by tools like ImageMagick or QuickTime or various Windows-based tools. See the dump image doc page for more details. E.g. this Imagemagick command would create a GIF file suitable for viewing in a browser.

```
% convert -loop 1 *.jpg foo.gif
```

There is also a COUPLE directory with examples of how to use SPPARKS as a library, either by itself or in tandem with another code or library. See the COUPLE/README file to get started.

10. Future plans

This section lists MC applications and features we are planning to add to SPPARKS. You can send an email to the developers if you are interested in any of these topics.

- off-lattice surface growth and diffusion
- chemical vapor deposition
- electromigration (Kristi Harris, UMBC)
- nanoporous aging (Greg Wagner, Sandia)
- pore migration (Veena Tikare, Sandia)

4. How-to discussions

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

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

4.1 Running multiple simulations from one input script

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

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

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

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

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

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

would run 2 independent simulations, one after the other.

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

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

```
jump in.runs
```

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

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

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

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

4.2 Coupling SPPARKS to other codes

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

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

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

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

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

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

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

Examples of driver codes that call SPPARKS as a library are included in the examples/COUPLE directory of the SPPARKS distribution; see examples/COUPLE/README for more details:

- simple: simple driver programs in C++ and C which invoke SPPARKS as a library (NOTE: not yet available)
- lammps_spparks: coupling of SPPARKS and LAMMPS, to couple a kinetic Monte Carlo model for grain growth using MD to calculate strain induced across grain boundaries

This section of the documentation describes how to build SPPARKS as a library. Once this is done, you can interface with SPPARKS either via C++, C, Fortran, or Python (or any other language that supports a vanilla C-like interface). For example, from C++ you could create one (or more) "instances" of 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. See Section_python of the manual for a description of the Python wrapper provided with SPPARKS that operates through the SPPARKS library interface.

The files src/library.cpp and library.h contain the C-style interface to SPPARKS. See Section_howto 3 of the manual for a description of the interface and how to extend it for your needs.

Note that the spparks_open() function that creates an instance of SPPARKS takes an MPI communicator as an argument. This means that instance of SPPARKS will run on the set of processors in the communicator. Thus 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. Or it might instantiate multiple instances of SPPARKS to perform different calculations.

4.3 Library interface to SPPARKS

As described in Section_start 4, SPPARKS can be built as a library, so that it can be called by another code, used in a coupled manner with other codes, or driven through a Python interface.

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

Library.cpp contains these 4 functions:

```
void spparks_open(int, char **, MPI_Comm, void **);
void spparks_close(void *);
void spparks_file(void *, char *);
char *spparks_command(void *, 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 in stand-alone mode 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 is used in subsequent library calls. 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. Or it might instantiate multiple instances of SPPARKS to perform different calculations.

The spparks_close() function is used to shut down an instance of 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 script or single command in an input script. Thus the calling code can read or generate a series of SPPARKS commands one line at a time and pass it thru the library interface to setup a problem and then run it, interleaving the spparks_command() calls with other calls to extract information from SPPARKS, perform its own operations, or call another code's library.

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

```
void *spparks_extract(void *, char *)
double *spparks_energy()
```

These can extract various global or per-site quantities from SPPARKS so that a driver application can access the values or even reset them. See the library.cpp file and its associated header file library.h for details.

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

1. Introduction

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

1.1 What is SPPARKS

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

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

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

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

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

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

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

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

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

1.2 SPPARKS features

These are three kinds of applications in SPPARKS:

- on-lattice
- off-lattice
- general

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

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

- diffusion model
- Ising model
- Potts model in many variants
- membrane model
- sintering model

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

- Metropolis atomic relaxation model

The set of general applications currently in SPPARKS are:

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

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

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

Pre- and post-processing:

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

1.3 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), or sometimes by request under the terms of the GNU Lesser General Public License (LGPL). This is often referred to as open-source distribution - see www.gnu.org or www.opensource.org for more details. The legal text of the GPL or LGPL is in the LICENSE file that is included in the SPPARKS distribution.

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

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

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

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

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

In the spirit of an open-source code, if you use SPPARKS for something useful or if you fix a bug or add a new feature or applicaton to the code, let us know. We would like to include your contribution in the released version of the code and/or advertise your success on our WWW page.

1.4 Acknowledgments and citations

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

The Authors page of the SPPARKS website lists the developers and their contact info, along with others who have contributed code and expertise to the developement of SPPARKS.

8. Modifying & extending SPPARKS

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

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

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

Most of the new features described in this section require you to write a new C++ derived class. Creating a new class requires 2 files, a source code file (*.cpp) and a header file (*.h). The derived class must provide certain methods to work as a new option. Depending on how different your new feature is compared to existing features, you can either derive from the base class itself, or from a derived class that already exists. Enabling SPPARKS to invoke the new class is as simple as adding two lines to the style_user.h file, in the same syntax as other SPPARKS classes are specified in the style.h file.

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

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

```
app_style foo 0.1 3.5
```

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

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

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

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

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

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

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

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

Application styles

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

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

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

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

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

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

Diagnostic styles

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

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

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

Input script commands

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

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

command	operations performed by the new command

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

Solve styles

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

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

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

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

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

6. Performance & scalability

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

9. Python interface to SPPARKS

This section describes how to build and use SPPARKS via a Python interface.

The SPPARKS distribution includes the file python/spparks.py which wraps the library interface to SPPARKS. This file makes it is possible to run SPPARKS, invoke SPPARKS commands or give it an input script, extract SPPARKS results, an modify internal SPPARKS variables, either from a Python script or interactively from a Python prompt. You can do the former in serial or parallel. Running Python interactively in parallel does not generally work, unless you have a package installed that extends your Python to enable multiple instances of Python to read what you type.

Python is a powerful scripting and programming language which can be used to wrap software like SPPARKS and other packages. It can be used to glue multiple pieces of software together, e.g. to run a coupled or multiscale model. See Section section of the manual for more ideas about coupling SPPARKS to other codes. See Section_start 4 about how to build SPPARKS as a library, and Section_howto 3 for a description of the library interface provided in src/library.cpp and src/library.h and how to extend it for your needs. As described below, that interface is what is exposed to Python. It is designed to be easy to add functions to. This can easily extend the Python inteface as well. See details below.

By using the Python interface, SPPARKS can also be coupled with a GUI or other visualization tools that display graphs or animations in real time as SPPARKS runs. Examples of such scripts may eventually be included in the python directory.

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

Before using SPPARKS from a Python script, you need to do two things. You need to build SPPARKS as a dynamic shared library, so it can be loaded by Python. And you need to tell Python how to find the library and the Python wrapper file python/spparks.py. Both these steps are discussed below. If you wish to run SPPARKS in parallel from Python, you also need to extend your Python with MPI. This is also discussed below.

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

9.1 Building SPPARKS as a shared library

Instructions on how to build SPPARKS as a shared library are given in Section_start 5. A shared library is one that is dynamically loadable, which is what Python requires. On Linux this is a library file that ends in ".so", not ".a".

From the src directory, type

```
make makeshlib
make -f Makefile.shlib foo
```

where foo is the machine target name, such as linux or g++ or serial. This should create the file libspparks_foo.so in the src directory, as well as a soft link libspparks.so, which is what the Python wrapper will load by default. Note that if you are building multiple machine versions of the shared library, the soft link is always set to the most recently built version.

If this fails, see Section_start 5 for more details, especially if your SPPARKS build uses auxiliary libraries like MPI which may not be built as shared libraries on your system.

9.2 Installing the Python wrapper into Python

For Python to invoke SPPARKS, there are 2 files it needs to know about:

- python/spparks.py
- src/libspparks.so

Spparks.py is the Python wrapper on the SPPARKS library interface. Libspparks.so is the shared SPPARKS library that Python loads, as described above.

You can insure Python can find these files in one of two ways:

- set two environment variables
- run the python/install.py script

If you set the paths to these files as environment variables, you only have to do it once. For the csh or tcsh shells, add something like this to your ~/.cshrc file, one line for each of the two files:

```
setenv PYTHONPATH $PYTHONPATH:/home/sjplimp/spparks/python
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/home/sjplimp/spparks/src
```

If you use the python/install.py script, you need to invoke it every time you rebuild SPPARKS (as a shared library) or make changes to the python/spparks.py file.

You can invoke install.py from the python directory as

```
% python install.py [libdir] [pydir]
```

The optional libdir is where to copy the SPPARKS shared library to; the default is /usr/local/lib. The optional pydir is where to copy the spparks.py file to; the default is the site-packages directory of the version of Python that is running the install script.

Note that libdir must be a location that is in your default LD_LIBRARY_PATH, like /usr/local/lib or /usr/lib. And pydir must be a location that Python looks in by default for imported modules, like its site-packages dir. If you want to copy these files to non-standard locations, such as within your own user space, you will need to set your PYTHONPATH and LD_LIBRARY_PATH environment variables accordingly, as above.

If the install.py script does not allow you to copy files into system directories, prefix the python command with "sudo". If you do this, make sure that the Python that root runs is the same as the Python you run. E.g. you may need to do something like

```
% sudo /usr/local/bin/python install.py [libdir] [pydir]
```

You can also invoke install.py from the make command in the src directory as

```
% make install-python
```

In this mode you cannot append optional arguments. Again, you may need to prefix this with "sudo". In this mode you cannot control which Python is invoked by root.

Note that if you want Python to be able to load different versions of the SPPARKS shared library (see this section below), you will need to manually copy files like libspparks_g++.so into the appropriate system directory. This is not needed if you set the LD_LIBRARY_PATH environment variable as described above.

9.3 Extending Python with MPI to run in parallel

If you wish to run SPPARKS in parallel from Python, you need to extend your Python with an interface to MPI. This also allows you to make MPI calls directly from Python in your script, if you desire.

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

These include

- pyMPI
- maroonmpi
- mpi4py
- myMPI
- Pypar

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

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

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

```
import numpy
```

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

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

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

To install Pypar (version pypar-2.1.4_94 as of Aug 2012), unpack it and from its "source" directory, type

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

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

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

```
import pypar
```

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

```
% mpirun -np 4 python test.py
```

where test.py contains the lines

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

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

IMPORTANT NOTE: To use Pypar and SPPARKS in parallel from Python, you must insure both are using the same version of MPI. If you only have one MPI installed on your system, this is not an issue, but it can be if you have multiple MPIs. Your SPPARKS build is explicit about which MPI it is using, since you specify the details in your lo-level src/MAKE/Makefile.foo file. Pypar uses the "mpicc" command to find information about the MPI it uses to build against. And it tries to load "libmpi.so" from the LD_LIBRARY_PATH. This may or may not find the MPI library that SPPARKS is using. If you have problems running both Pypar and SPPARKS together, this is an issue you may need to address, e.g. by moving other MPI installations so that Pypar finds the right one.

9.4 Testing the Python-SPPARKS interface

To test if SPPARKS is callable from Python, launch Python interactively and type:

```
>>> from spparks import spparks
>>> spk = spparks()
```

If you get no errors, you're ready to use SPPARKS from Python. If the 2nd command fails, the most common error to see is

```
OSError: Could not load SPPARKS dynamic library
```

which means Python was unable to load the SPPARKS shared library. This typically occurs if the system can't find the SPPARKS shared library or one of the auxiliary shared libraries it depends on, or if something about the library is incompatible with your Python. The error message should give you an indication of what went wrong.

You can also test the load directly in Python as follows, without first importing from the spparks.py file:

```
>>> from ctypes import CDLL
>>> CDLL("libspparks.so")
```

If an error occurs, carefully go thru the steps in Section_start 5 and above about building a shared library and about insuring Python can find the necessary two files it needs.

Test SPPARKS and Python in serial:

To run a SPPARKS test in serial, type these lines into Python interactively from the examples/ising directory:

```
>>> from spparks import spparks
>>> spk = spparks()
>>> spk.file("in.ising")
```

Or put the same lines in the file test.py and run it as

```
% python test.py
```

Either way, you should see the results of running the in.ising example on a single processor appear on the screen, the same as if you had typed something like:

```
spk_g++ <in.ising
```

Test SPPARKS and Python in parallel:

To run SPPARKS in parallel, assuming you have installed the Pypar package as discussed above, create a test.py file containing these lines:

```
import pypar
from spparks import spparks
spk = spparks()
spk.file("in.ising")
print "Proc %d out of %d procs has" % (pypar.rank(),pypar.size()),spk
pypar.finalize()
```

You can then run it in parallel as:

```
% mpirun -np 4 python test.py
```

and you should see the same output as if you had typed

```
% mpirun -np 4 spk_g++ <in.ising
```

Note that if you leave out the 3 lines from test.py that specify Pypar commands you will instantiate and run SPPARKS independently on each of the P processors specified in the mpirun command. In this case you should get 4 sets of output, each showing that a SPPARKS run was made on a single processor, instead of one set of output showing that SPPARKS ran on 4 processors. If the 1-processor outputs occur, it means that Pypar is not working correctly.

Also note that once you import the PyPar module, Pypar initializes MPI for you, and you can use MPI calls directly in your Python script, as described in the Pypar documentation. The last line of your Python script should be pypar.finalize(), to insure MPI is shut down correctly.

Running Python scripts:

Note that any Python script (not just for SPPARKS) can be invoked in one of several ways:

```
% python foo.script
% python -i foo.script
% foo.script
```

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

```
#!/usr/local/bin/python
#!/usr/local/bin/python -i
```

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

```
% chmod +x foo.script
```

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

9.5 Using SPPARKS from Python

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

```
from spparks import spparks
```

These are the methods defined by the spparks module. If you look at the file src/library.cpp you will see that they correspond one-to-one with calls you can make to the SPPARKS library from a C++ or C or Fortran program.

```
spk = spparks()           # create a SPPARKS object using the default libspparks.so library
spk = spparks("g++")      # create a SPPARKS object using the libspparks_g++.so library
spk = spparks("",list)    # ditto, with command-line args, e.g. list = ["-echo","screen"]
spk = spparks("g++",list)

spk.close()              # destroy a SPPARKS object

spk.file(file)           # run an entire input script, file = "in.lj"
spk.command(cmd)         # invoke a single SPPARKS command, cmd = "run 100.0"

xlo = spk.extract(name,type)  # extract a global quantity
                              # name = "boxxlo", "nlocal", "id", "xyz", "site", iarray2", "da
                              # type = 0 = int
                              #        1 = int vector
                              #        2 = int array
                              #        3 = double
                              #        4 = double vector
```

```
                                #         5 = double array

eng = spk.energy()       # query current energy of system
```

IMPORTANT NOTE: Currently, the creation of a SPPARKS object from within spparks.py does not take an MPI communicator as an argument. There should be a way to do this, so that the SPPARKS instance runs on a subset of processors if desired, but I don't know how to do it from Pypar. So for now, it runs with MPI_COMM_WORLD, which is all the processors. If someone figures out how to do this with one or more of the Python wrappers for MPI, like Pypar, please let us know and we will amend these doc pages.

Note that you can create multiple SPPARKS objects in your Python script, and coordinate and run multiple simulations, e.g.

```
from spparks import spparks
spk1 = spparks()
spk2 = spparks()
spk1.file("in.file1")
spk2.file("in.file2")
```

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

The extract() method returns values or pointers to data structures internal to SPPARKS. See the src/app.cpp file and its extract() method for a list of what is recognized as "name" arguments. Other values could easily be added.

For example, "boxxlo" returns the lower x-bound of the simulation box. "Nlocal" and "nglobal" return the number of lattice sites owned by a proc or the total # of lattice sites in the simulation. "Xyz" returns the Nx3 array of lattice site coordinates. "Site" and "iarrayN" and "darrayN" return a vector of integer or floating-point per-site values.

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

- Add a new interface function to src/library.cpp and src/library.h.
- Rebuild SPPARKS as a shared library.
- Add a wrapper method to python/spparks.py for this interface function.
- You should now be able to invoke the new interface function from a Python script. Isn't ctypes amazing?

9.6 Example Python scripts that use SPPARKS

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

trivial.py	read/run a SPPARKS input script thru Python
demo.py	invoke various SPPARKS library interface routines

See the python/README file for instructions on how to run them and the source code for individual scripts for comments about what they do.

2. Getting Started

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

2.1 What's in the SPPARKS distribution

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

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

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

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

2.2 Making SPPARKS

This section has the following sub-sections:

Read this first:

Building SPPARKS can be non-trivial. You will likely need to edit a makefile, there are compiler options, additional libraries can be used (MPI, JPEG), 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, say *serial* or *mpi* or *linux*, then type one of the commands:

```
make serial
make mpi
gmake linux
```

Try the "serial" and "mpi" targets first, since they are generic and should typically work on any machine, assuming you have the GNU g++ compiler (for the serial version) and MPI installed (for the mpi version).

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_serial or spk_mpi is produced, you're done; it's your lucky day.

IMPORTANT NOTE: You need a C++ compiler that is C++11 compliant to build SPPARKS. Almost all current C++ compilers are; you just need to use a -std=c++11 flag when compiling, as in the src/MAKE/Makefile.machine files provided with SPPARKS.

Common 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.

(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) The "compiler/linker settings" section lists compiler and linker settings for your C++ compiler, including path and optimization flags. You can use g++, the open-source GNU compiler, which is available on all Unix systems. You can also use mpicc which will typically be available if MPI is installed on your system, though you should check which actual compiler it wraps. You can also point to a specific compiler; for example see MAKE/Makefile.spencer.gnu where an environment variable MPI_HOME is used to specify path to mpicxx and mpicc compilers.

Vendor compilers often produce faster code. On boxes with Intel CPUs, we suggest using the commercial Intel icc compiler, which can be downloaded from Intel's compiler site.

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

The DEPFLAGS setting is what triggers the C++ compiler to create a dependency list for a source file. This speeds re-compilation when source (*.cpp) or header (*.h) files are edited. Some compilers do not support dependency file creation, or may use a different switch than -D. GNU g++ works with -D. If your compiler can't create dependency files (a long list of errors involving *.d files), then you'll need to create a Makefile.foo patterned after Makefile.storm, which uses different rules that do not involve dependency files.

(3) The "system-specific settings" section has 3 parts.

(3.a) The SPK_INC variable is used to include options that turn on system-dependent ifdefs within the SPPARKS code. The settings that are currently recogized are:

- -DSPPARKS_GZIP
- -DSPPARKS_JPEG
- -DSPPARKS_SMALLBIG
- -DSPPARKS_BIGBIG
- -DSPPARKS_SMALLSMALL

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

If you use -DSPPARKS_JPEG, the dump image command will be able to write out JPEG image files. If not, it will only be able to write out text-based PPM image files. For JPEG files, you must also link SPPARKS with a JPEG library. See section (3.c) below for more details on this.

Use at most one of the -DSPPARKS_SMALLBIG, -DSPPARKS_BIGBIG, -DSPPARKS_SMALLSMALL settings. The default is -DSPPARKS_SMALLBIG. These settings refer to use of 4-byte (small) vs 8-byte (big) integers within SPPARKS, as specified in src/spktype.h. The only reason to use the BIGBIG setting is to enable simulation of systems with more than 2 billion sites. Normally, the only reason to use SMALLSMALL is if your machine does not support 64-bit integers. See the Additional build tips section below for more details on these settings.

(3.b) The 3 MPI variables are used to specify an MPI library to build SPPARKS with.

If you want SPPARKS to run in parallel, you must have an MPI library installed on your platform. If you use an MPI-wrapped compiler, such as "mpicc" to build SPPARKS, you can probably leave these 3 variables blank. If you do not use "mpicc" as your compiler/linker, then you need to specify where the mpi.h file (MPI_INC) and the MPI library (MPI_PATH) is found and its name (MPI_LIB).

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

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 a true MPI library installed on your system. See the Makefile.serial file for how to specify the 3 MPI variables. 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 this 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.

(3.c) The 3 JPG variables are used to specify a JPEG library which SPPARKS uses when writing a JPEG file via the dump image command. These can be left blank if you are not using the -DSPPARKS_JPEG switch discussed above in section (3.a).

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

That's it. Once you have a correct Makefile.foo and you have pre-built any other libraries it will use (e.g. MPI, JPEG), all you need to do from the src directory is type one of these 2 commands:

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) Changing the SPPARKS size limits via -DSPPARKS_SMALLBIG or -DSPPARKS_BIGBIG or -DSPPARKS_SMALLSMALL

As explained above, any of these 3 settings can be specified on the SPK_INC line in your low-level src/MAKE/Makefile.foo.

The default is -DSPPARKS_SMALLBIG which allows for systems with up to 2^31 sites (about 2 billion). This is because the site IDs are stored in 32-bit integers.

To allow for larger systems, compile with -DSPPARKS_BIGBIG. This stores site IDs in 64-bit integers. This enables systems with up to 2^63 sites (about 9e18).

If your system does not support 8-byte integers, you will need to compile with the -DSPPARKS_SMALLSMALL setting. This will restrict the total number of sites to 2^31 (about 2 billion), as well as store some simulation statistics in 4-byte integers.

Note that in src/lmptype.h there are definitions of all these data types as well as the MPI data types associated with them. The MPI types need to be consistent with the associated C data types, or else SPPARKS will generate a run-time error. As far as we know, the settings defined in src/spktype.h are portable and work on every current system.

In all cases, the size of problem that can be run on a per-processor basis is limited by 4-byte integer storage to 2^31 sites per processor (about 2 billion). This should not normally be a limitation since such a problem would have a huge per-processor memory and would run very slowly in terms of CPU secs per Monte Carlo interation.

Building for a Mac:

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

Building for Windows:

SPPARKS is just C++ with MPI calls, so it should be possible to build it for a Windows box, either using a Linux installation such as cygwin (see src/MAKE/Makefile.cygwin), or importing the source files into Visual Studio C++ and building it there. For the latter you are on your own. The SPPARKS developers do not use Windows. But if you figure out how to do it, or create a Visual Studio project that works, please let us know, and we can release the instructions/files for how to do this as part of SPPARKS.

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".

Currently there is only one optional package: STITCH. It is dicussed more below.

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.

2.3.1 STITCH package

The STITCH package allows SPPARKS to use the Stitch library for I/O, which is included in the SPPARKS distribution in lib/stitch. At some point the Stitch library will have its own website and will also be downloadable

there.

Stitch is an efficient I/O API and database format with a native python interface. *Stitch* files can read in to start a simulation and/or output during a simulation. A novel aspect of *stitch* is that it enables out-of-core computations by building a simulation domain analogously to the way an additive manufactured (AM) part is built. It merges outputs written over time to efficiently construct a much larger simulation domain that would otherwise be impossible to model in one simulation. *Stitching* workflows can be created to perform multiple SPPARKS simulations representing an additive manufacturing process; such simulations can produce huge numbers of lattice sites representing an entire AM build that would otherwise be impossible to simulate due to length scale and computational resource limitations. *Stitch* is intended and primarily focused on microstructural evolution simulations such as welding and additive manufacturing but other applications may be possible.

Building SPPARKS with the STITCH package enables these commands to use stitch-related options:

- dump stitch
- set stitch
- reset_time

See the am_path and stitch sub-directories in the examples directory for models and scripts which use the Stitch library.

You can build SPPARKS with *stitch* support in one of 3 ways.

(1) From the src directory using make

```
% cd spparks/src
% make lib-stitch args="-b"   # build the Stitch library and set links to it
% make yes-stitch             # install the STITCH package
% make mpi                    # or whichever machine target you wish
```

(2) From the lib directory using Install.py

```
% cd spparks/lib
% python Install.py -b        # build the Stitch library and set links to it
% cd spparks/src
% make yes-stitch             # install the STITCH package
% make mpi                    # or whichever machine target you wish
```

(3) Manual build of the Stitch library you have downloaded to your system

```
% cd $STITCHDIR                # STITCHDIR = the Stitch library directory
% make                         # build Stitch with default Makefile
% make -f Makefile.custom      # build Stitch with custom Makefile
% cd spparks/lib/stitch
% ln -s $STITCHDIR liblink     # set two links in SPPARKS lib/stitch
% ln -s $STITCHDIR includelink
% cd spparks/src
% make yes-stitch             # install the STITCH package
% make mpi                    # or whichever machine target you wish
```

To un-install the STITCH package from SPPARKS, do the following:

```
% cd spparks/src
% make no-stitch     # un-install the STITCH package files
% make mpi           # re-build SPPARKS w/out the STITCH package
```

2.4 Building SPPARKS as a library

SPPARKS can be built as either a static or shared library, which can then be called from another application or a scripting language. See this section for more info on coupling SPPARKS to other codes. See this section for more info on wrapping and running SPPARKS from Python.

Static library:

To build SPPARKS as a static library (*.a file on Linux), type

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

where foo is the machine name. This kind of library is typically used to statically link a driver application to SPPARKS, so that you can insure all dependencies are satisfied at compile time. Note that inclusion or exclusion of any desired optional packages should be done before typing "make makelib". The first "make" command will create a current Makefile.lib with all the file names in your src dir. The second "make" command will use it to build SPPARKS as a static library, using the ARCHIVE and ARFLAGS settings in src/MAKE/Makefile.foo. The build will create the file libspparks_foo.a which another application can link to.

Shared library:

To build SPPARKS as a shared library (*.so file on Linux), which can be dynamically loaded, e.g. from Python, type

```
make makeshlib
make -f Makefile.shlib foo
```

where foo is the machine name. This kind of library is required when wrapping SPPARKS with Python; see Section_python for details. Again, note that inclusion or exclusion of any desired optional packages should be done before typing "make makelib". The first "make" command will create a current Makefile.shlib with all the file names in your src dir. The second "make" command will use it to build SPPARKS as a shared library, using the SHFLAGS and SHLIBFLAGS settings in src/MAKE/Makefile.foo. The build will create the file libspparks_foo.so which another application can link to dyamically. It will also create a soft link libspparks.so, which the Python wrapper uses by default.

Note that for a shared library to be usable by a calling program, all the auxiliary libraries it depends on must also exist as shared libraries. This will be the case for libraries included with SPPARKS, such as the dummy MPI library in src/STUBS since they are always built as shared libraries with the -fPIC switch. However, if a library like MPI does not exist as a shared library, the second make command will generate an error. This means you will need to install a shared library version of the package. The build instructions for the library should tell you how to do this.

As an example, here is how to build and install the MPICH library, a popular open-source version of MPI, distributed by Argonne National Labs, as a shared library in the default /usr/local/lib location:

```
./configure --enable-shared
make
make install
```

You may need to use "sudo make install" in place of the last line if you do not have write privileges for /usr/local/lib. The end result should be the file /usr/local/lib/libmpich.so.

Additional requirement for using a shared library:

The operating system finds shared libraries to load at run-time using the environment variable LD_LIBRARY_PATH. So you may wish to copy the file src/libspparks.so or src/libspparks_g++.so (for example) to a place the system can find it by default, such as /usr/local/lib, or you may wish to add the SPPARKS src directory to LD_LIBRARY_PATH, so that the current version of the shared library is always available to programs that use it.

For the csh or tcsh shells, you would add something like this to your ~/.cshrc file:

```
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/home/sjplimp/spparks/src
```

Calling the SPPARKS library:

Either flavor of library (static or shared0 allows one or more SPPARKS objects to be instantiated from the calling program.

When used from a C++ program, all of SPPARKS is wrapped in a SPPARKS_NS namespace; you can safely use any of its classes and methods from within the calling code, as needed.

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

See the sample codes in examples/COUPLE/simple for examples of C++ and C and Fortran codes that invoke SPPARKS thru its library interface. There are other examples as well in the COUPLE directory which are discussed in Section_howto 2 of the manual. See Section_python of the manual for a description of the Python wrapper provided with SPPARKS that operates through the SPPARKS library interface.

The files src/library.cpp and library.h define the C-style API for using SPPARKS as a library. See Section_howto 3 of the manual for a description of the interface and how to extend it for your needs.

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 a section below. 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.

2.7 SPPARKS screen output

As SPPARKS reads an input script, it prints information to both the screen and a log file about significant actions it takes to setup a simulation. When the simulation is ready to begin, SPPARKS performs various initializations and prints the amount of memory (in MBytes per processor) that the simulation requires. An example output is shown here, for the examples/in.potts script run on 4 processors.

```
SPPARKS (11 Dec 2015)
Created box = (0 0 0) to (20 20 20)
  1 by 2 by 2 processor grid
Creating sites ...
  8000 sites
  8000 sites have 26 neighbors
Setting site values ...
  8000 settings made for site
Setting up run ...
Memory usage per processor = 4.375 Mbytes
```

During the run itself, statistical information is printed periodically, for every delta of simulation time, as specified by the stats commmand. When the run concludes, SPPARKS prints final statistical info and a total run time for the simulation.

```
      Time    Naccept    Nreject    Nsweeps         CPU     Energy
         0          0          0          0           0     205912
     10.01      88437    7919563       1001       0.195      72506
        20      94828   15905172       2000       0.379      57038
        30      98345   23901655       3000       0.565      49948
        40     101449   31898551       4000       0.749      44316
     50.01     103978   39904022       5001       0.933      39334
     60.01     105578   47902422       6001        1.12      36902
     70.01     106938   55901062       7001         1.3      34428
        80     108491   63891509       8000        1.49      31668
        90     110211   71889789       9000        1.67      27994
```

```
       100     112074   79887926      10000       1.86      21894
Loop time of 1.86084 on 4 procs
```

It then appends statistics about the breakdown of CPU time for the simulation.

```
Solve time (%) = 1.52001 (81.6842)
Update time (%) = 0 (0)
Comm  time (%) = 0.245275 (13.1809)
Outpt time (%) = 0.0892967 (4.79874)
App   time (%) = 0 (0)
Other time (%) = 0.00625533 (0.336157)
```

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.

Addtional scripts below are distributed with spparks under the tools directory.

- potts_quaternion/cpp_quaternion.py: enables reading spparks quaternion header files
- potts_quaternion/plot_cubic_symmetry_histograms.py: verification plots for disorientation distribution of randomly oriented cubic structures
- potts_quaternion/plot_hcp_symmetry_histograms.py: verification plots for disorientation distribution of randomly oriented hcp structures

add_reaction command

Syntax:

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

- ID = string identifier for the reaction
- 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 1 A B 1.0e10 C
add_reaction Dreact 1.0 d
add_reaction myReact b2 1.0e-10 c3 d4 e3
```

Description:

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

The ID is simply a unique string (alphanumeric characters, dashes, underscores, etc) which helps identify the reaction in an input script listing.

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

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

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

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

Restrictions:

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

Related commands:

app_style chemistry, add_species

Default: none

add_species command

Syntax:

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

- name1,name2 = ID strings for different species

Examples:

```
add_species kinase
add_species NFkB kinase2 NFkB-IKK
```

Description:

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

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

Restrictions:

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

Related commands:

app_style chemistry, add_reaction, count

Default: none

am build command

Syntax:

```
am build start z num_layers N
```

- start = optional key word
- z = specifies elevation in SPPARKS sites of first layer
- num_layers = optional keyword
- N = specifies number of layers for this build simulation

Example 1:

```
am pass 1 dir X speed 9 hatch 75

am cartesian_layer 1 start LL pass_id 1 thickness 10 offset -100. 0.0

am build start 10 num_layers 2
```

Example 2:

```
am pass 1 dir X speed 9 hatch 75

am pass 1 dir Y speed 9 hatch 75

am cartesian_layer 1 start LL pass_id 1 thickness 10 offset -100. 0.0

am cartesian_layer 2 start LR pass_id 2 thickness 1 offset 0.0 -100.0

am build start 10 num_layers 4
```

Description:

This is an optional command used by am/ellipsoid and potts/am/weld applications to specify multilayer build simulations. The command allows for re-use of layer specifications and implicitly creates a pattern of layers. The build pattern is implied by the order and number of layers in the input script. As is conventional, the build proceeds in the z-direction according to specified layer thicknesses. The *am build start* parameter specifies the top surface of build plane; its important to specify this parameter if the spparks domain is thicker than a build layer otherwise the default value will be *zhi* taken from region box which is probably not desired. Once all layers have been built/simulated, the pattern repeats, cycling through the layers again and again until *num_layers* have been simulated. The *am build* command allows for defining an arbitrary number of layers and patterns.

This command is mostly intended for SPPARKS simulations that do not use Stitch IO; nonetheless, this command can be used with Stitch IO. Simulations using Stitch IO would normally proceed layer-by-layer using only one layer in any particular simulation. If this command is omitted then the pattern of layers in the input script is only simulated once.

In Example 1, one layer is simulated. Because *thickness t=10*, *am build start z=10* is specified.

In Example 2 above, 2 layers are defined but 4 layers are simulated; layers are alternately rastered in *X* then *Y* directions starting at the LL corner and alternately the LR corner. As in Example 1, because first layer *thickness t=10*, the *start* value is set at *am build start z=10*.

Restrictions:

This command can only be used as part of the app_style am/ellipsoid app_style potts/am/weld or applications.

Related commands:

am pass, am path, am cartesian_layer am path

Default:

These are the option defaults:

- start z = the z-component of the SPPARKS region box
- num_layers = the number of layers in the input script

am cartesian_layer command

Syntax:

```
am cartesian_layer layer_id start location pass_id p_id thickness t offset x,y serpentine swit
```

- layer_id = integer identification number for this cartesian_layer
- start = required keyword
- location = LL or LR or UR or UL
- pass_id = required keyword
- p_id = integer ID of the pass to use for this layer
- thickness = required keyword
- t = thickness in sites of layer
- offset = optional keyword and x,y values
- x,y = optional relative offset (shift) in x-y plane of raster starting position
- serpentine = optional keyword and switch value
- switch = 0 for parallel, 1 for anti-parallel of consecutive scans

Examples:

```
# Example 1
am pass 1 dir X speed 9 hatch 75
am cartesian_layer 1 start LL pass_id 1 thickness 1 offset -100. 0.0

# Example 2
am pass 2 dir Y speed 9 hatch 75
am cartesian_layer 1 start LR pass_id 2 thickness 1 offset 0.0 -100.
```

Description:

This command is used by am/ellipsoid, and potts/am/weld applications to specify raster patterns on cartesian build layers. Multiple cartesian_layers can be defined in a single input file, using unique id values.

The *start location*, references the lower left LL, lower right LR, upper right UR, upper left UL of the rectangular domain; *location* must be one of *LL,LR,UR,UL*. The extent of the rectangular domain created by the raster pattern is specified elsewhere (region) using *region box* in the standard way.

The *pass_id p_id* specified must reference an am pass from earlier in the input script.

The build layer *thickness* is specified in units of sites.

Specified in units of sites, *offset* specifies the raster scan starting position relative to *start location*.

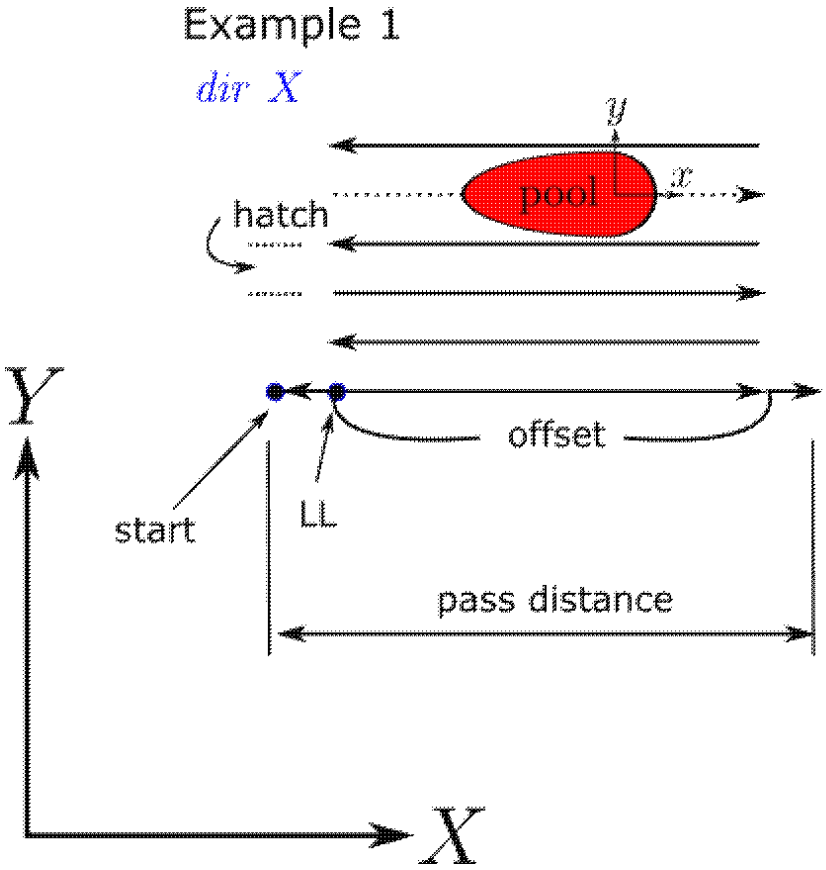
By default, scan patterns are anti-parallel *serpentine*. To turn off the default scan pattern add the optional switch *serpentine 0*.

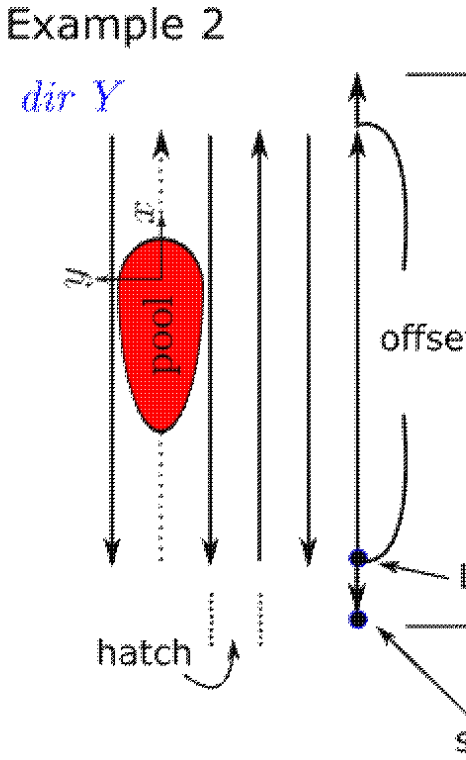
The initial heading of the scan pattern depends upon *start* and the value of *pass dir* (see am pass); also see examples below.

In the examples, two unique layers are created. *offset* option is used to extend starting and ending points of pass, effectively lengthening the pass; although offset consists of *x,y* values, it is primarily intended to change starting

and ending points along *dir*, ie only one component of *offset* is non-zero; although *offset* defaults to *x=0,y=0*, if one of these values must be non-zero, then *0.0* must be given for the other value.

serpentine 0 must be given to turn *serpentine* off.





Restrictions:

This command can only be used as part of apps am/ellipsoid or potts/am/weld.

Related commands:

am_pass, am_path, am_build

Default:

The defaults are offset x = 0.0, y = 0.0 and serpentine switch = 0.0.

am pass command

Syntax:

```
am pass id dir d speed V hatch h overhatch oh
```

- id = integer identification number for this pass
- dir = required keyword
- d = *X* or *Y* which specifies direction of the cartesian pass
- speed = required keyword
- V = scan speed of the pass in sites/Monte-Carlo-sweep
- hatch = required keyword
- h = hatch spacing distance
- overhatch = optional keyword
- oh = perpindicular distance to pass dir; expands domain size

Examples:

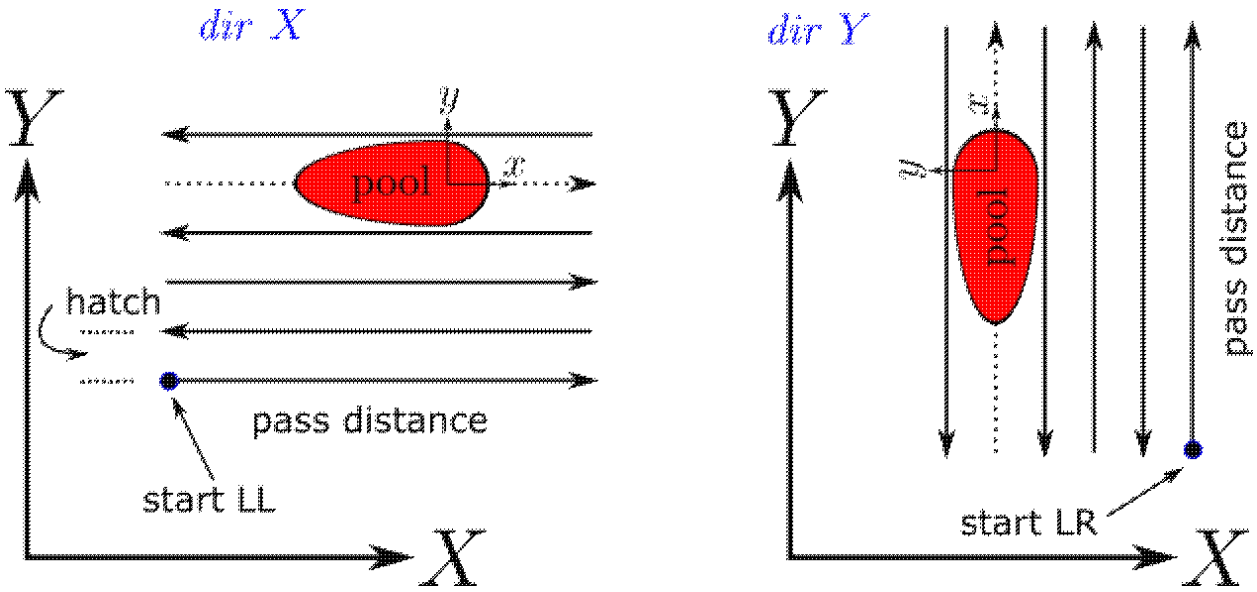
```
am pass 1 dir X speed 9 hatch 75
am pass 2 dir Y speed 10 hatch 50 overhatch 25
```

Description:

This command is used by am/ellipsoid, and potts/am_weld to specify raster scan patterns for rectangular domains. Multiple passes can be defined in a single input file, each using unique id values. The *am pass* command is a required subelement of am cartesian_layer that specifies a cartesian build layer in the *x,y* plane. The initial heading, left or right for *dir=X* or up or down for *dir=Y* depends upon the starting location specified in the am cartesian_layer command. By default, scan lines are *serpentine* but this can be turned off in the am cartesian_layer command.

The *pass* distance and number of *hatch* lines are controlled by the *X,Y* extent of the domain specified using the region command, which must be a box; multiple passes, seperated by *hatch* spacing are invoked until the domain dimension perpindicular to the scan *dir* is exhausted. The optional *overhatch* value can be used to cause additional scan lines. The melt pool has its own local coordinate system *x,y* so that the pool axis *x* is always oriented along *dir d*.

The example commands above define two passes, each with a different *dir*, *speed*, *hatch* and optional *overhatch*.



Restrictions:

This rastering command can be used with app_style potts/am/weld or app_style am/ellipsoid.

Related commands:

am_cartesian_layer

Default:

The default for overhatch = 0.0.

am path command

Syntax:

```
am path id start x0 y0 end x1 y1 speed V
```

- id = integer identification number for this path
- start = required keyword
- x0 y0 = starting point x-y plane for straight line path
- end = required keyword
- x1 y1 = end point x-y plane for straight line path
- speed = required keyword
- V = scan speed of the pass in sites/Monte-Carlo-sweep

Example

This snippet is taken from SPPARKS repository: *examples/am_path/path_raster_a*

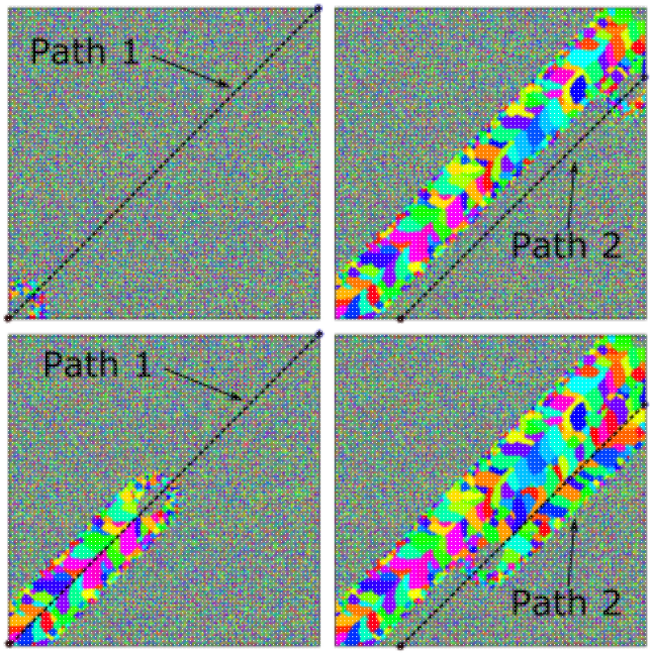
```
# Convenience; define set of points to be used in path commands
variable X0 equal         0.0
variable Y0 equal         0.0
variable X1 equal       500.0
variable Y1 equal       500.0
variable X2 equal       106.1
variable Y2 equal         0.0
variable X3 equal       500.0
variable Y3 equal       393.9

am path 1 start $X0 $Y0 end $X1 $Y1 speed 9
am path 2 start $X3 $Y3 end $X2 $Y2 speed 9
am path_layer 1 num_paths 2 path_ids 1 2 thickness 1
```

Description:

This command is used by am/ellipsoid, and potts/am_weld to specify raster scans in a very general way. This *am path* command specifies a line in the x-y plane using *start x,y* and *end x,y* points; path scan *speed V* is also part of the specification. Any number of *am path*s can be specified in a script; an *am path* can be associated with multiple *am path_layer*s. The direction of travel is implied by starting and ending points.

The example commands above define two paths both each of which are associated with *am path_layer 1*. See depiction of simulation below.



Restrictions:

This rastering command can be used with app_style potts/am/weld or app_style am/ellipsoid.

Related commands:

The *am path* is a required element of am_path_layer

Default: none

am path_layer command

Syntax:

```
am path_layer layer_id num_paths N path_ids (tuple paths_ids) thickness t
```

- layer_id = integer identification number for this path_layer
- num_paths = required keyword
- N = integer number of paths specified on this layer
- path_ids = required keyword
- (tuple path_ids) = tuple array of path_ids corresponding *am path* specifications elsewhere in script
- thickness = required keyword
- t = thickness in sites of layer

Examples

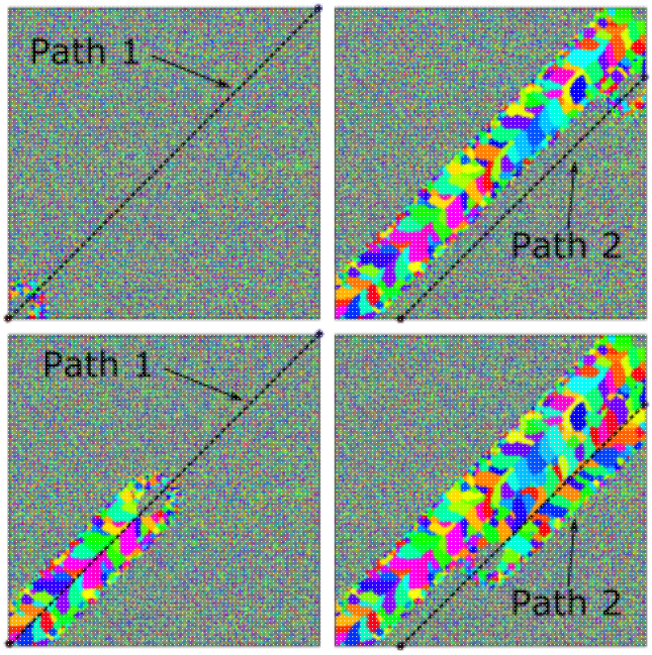
```
variable X0 equal         0.0
variable Y0 equal         0.0
variable X1 equal       500.0
variable Y1 equal       500.0
variable X2 equal       106.1
variable Y2 equal         0.0
variable X3 equal       500.0
variable Y3 equal       393.9
am path 1 start $X0 $Y0 end $X1 $Y1 speed 9
am path 2 start $X3 $Y3 end $X2 $Y2 speed 9
am path_layer 1 num_paths 2 path_ids 1 2 thickness 1
```

Description:

This command is used by am/ellipsoid, and potts/am/weld applications to specify raster patterns on path build layers. Multiple path_layers can be defined in a single input file, using unique id values. Although simple build simulations can be conducted using this command by manually writing scripts that use *am path_layer*, it is expected that this command will generally be automatically created for the purpose of handling more complex geometries.

The *num_paths* and *path_ids* keywords are used to create the layer with *thickness t*. The build layer *thickness* is specified in units of sites.

In example above, two unique *am path*s are created; these paths are referenced in the *am path_layer* command. This example is also illustrated for am_path,



Restrictions:

This command can only be used as part of apps am/ellipsoid or potts/am/weld applications.

Related commands:

am_pass, am_path, am_build

Default: none

am pathgen command

Syntax:

```
am pathgen outfile "filename" num_layers "N" zstart "Z" width_haz "H" melt_depth D depth_haz
```

- outfile = required keyword
- filename = name of output file containing CV and raster path information
- num_layers = required keyword
- N = number of layers paths are generated for
- zstart = required keyword
- Z = starting elevation of build
- width_haz = required keyword
- H = width of heat effected zone; always greater than pool width
- melt_depth = required keyword
- D = melt depth; generally greater than layer thickness and smaller than 2 layers thick
- depth_haz = required keyword
- DH = depth of heat effected zone; always greater than melt_depth

Examples:

Taken from examples/stitch/stitching_rectangular_domain.

```
####################################
# Variables defined for convenience
# WIDTH_HAZ
variable WIDTH_HAZ equal 13
#
# MELT_DEPTH
variable MELT_DEPTH equal 4
#
# DEPTH_HAZ
variable DEPTH_HAZ equal 5
#
# V: scan speed
variable V equal 14.0
#
# HATCH: hatch spacing
variable HATCH equal 5.0
#
# LAYER_THICKNESS:
variable LAYER_THICKNESS equal 3
#
# OUT: output filename
variable OUT world pathgen.dat
####################################

#########################################################
# This example uses the following larger intended domain
region       box block  0 100 0 280 0 48
#########################################################

#############################################################
# Additional commands defining hatch pattern, cartesian layers, etc
am pass 1 dir X speed $V hatch $HATCH
am pass 2 dir Y speed $V hatch $HATCH
```

```
am cartesian_layer 1 start LL pass_id 1 thickness $LAYER_THICKNESS offset -80.0 0.0 serpentine
am cartesian_layer 2 start UL pass_id 2 thickness $LAYER_THICKNESS offset 0.0 80.0 serpentine
am cartesian_layer 3 start UR pass_id 1 thickness $LAYER_THICKNESS offset 80.0 0.0 serpentine
am cartesian_layer 4 start LR pass_id 2 thickness $LAYER_THICKNESS offset 0.0 -80.0 serpentine
##########################################################

Example 1:
pathgen outfile $OUT num_layers 6 zstart 0 width_haz $WIDTH_HAZ melt_depth $MELT_DEPTH depth_h
```

Description:

This command is used by potts/am/path/gen for auto generation of path information on rectangular layers. Output from this command can be used to assemble very large simulations of AM microstructures using a sequence of significantly smaller simulations. The sequence of smaller simulations are conducted on a series of CVs calculated using SPPARKS on the basis of the larger intended domain size as specified using the standard region command.

The *outfile* parameter specifies name of file where path information is written. This file is subsequently read line by line using a python or bash script to orchestrate the ordered series of smaller simulations.

num_layers parameter specifies how many layers output path information will be generated for. This parameter allows for re-use of layer specifications and implicitly creates a pattern of layers. The build pattern is implied by the order and number of layers in the input script; if *num_layers* is greater than number of cartesian layers provided in script then the input cartesian layers are used as a pattern repeated as necessary to create number of layers. Once path information has been generated for all layers specified, the pattern repeats, cycling through the layers again and again until *num_layers* have been processed.

As is conventional, the build proceeds in the z-direction according to specified layer thicknesses. The *zstart* parameter specifies starting z elevation of build plane surface.

On the basis of specified *am pass* and heat effected zone parameters, a sequence of computational volumes (CV) are created. The *width_haz* parameter sizes the width of the CV; the length of the CV is collected from *am pass* and domain size information while depth of the CV is specified with the *depth_haz* parameter. Microstructure simulations are conducted on the smaller incremental sequence of CVs. The *depth_haz* parameter only makes sense when its greater than *melt_depth*. The *melt_depth* parameter is not used directly by the path generator but is passed on to the AM model for simulating microstructures; for convenience it is included here to allow for parametric studies on melt_depth and depth_haz.

Restrictions:

This command is only used by the potts/am/path/gen app in conjunction with the region, create_box, am pass, and am cartesian_layer, commands. If the am pathgen command is missing, the potts/am/path/gen app will run but path information will not be generated.

Related commands:

potts/am/path/gen region create_box am_pass am_cartesian_layer

Default:

None

app_style am/ellipsoid command

Syntax:

```
app_style am/ellipsoid nspins spot_width melt_tail_length melt_depth cap_height HAZ tail_HAZ d
```

- am/ellipsoid = application style name
- nspins = number of possible spins
- spot_width = maximum width of the melt pool
- melt_tail_length = maximum length of the melt pool trailing the melt spot
- melt_depth = maximum depth of the melt pool
- cap_height = maximum length of the melt pool leading the melt spot
- HAZ = width of the heat affected zone (haz) surrounding the melt pool (must be larger than width)
- tail_HAZ = Length of the haz trailing the meltpool (must be larger than tail_length)
- depth_HAZ = depth of the heat affect zone (haz) below the melt pool (must be larger than depth)
- cap_HAZ = Length of haz leading the melt pool (must be larger than cap_length)
- exp_factor = Coefficient that controls the rate of exponential decay of the haz mobility gradient

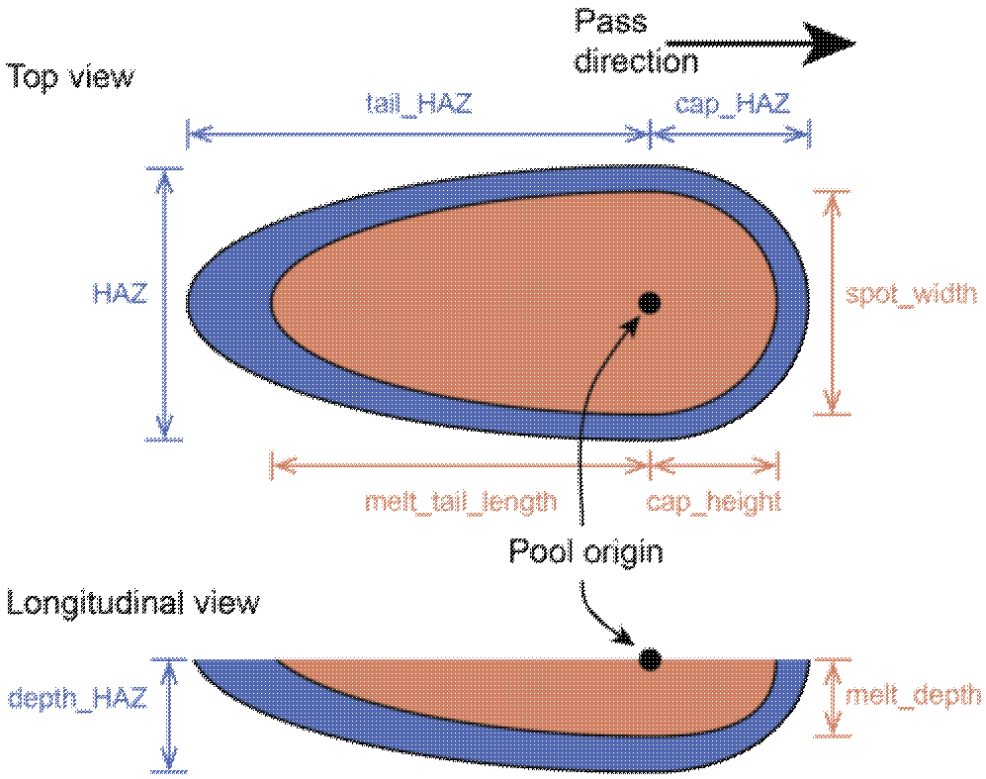
Examples:

```
app_style am/ellipsoid 1000 30 40 20 5 50 60 30 7 0.1
```

Description:

This is an on-lattice application derived from the app_style potts/neighonly application that simulates the rastering of a molten pool and its accompanying heat-affected zone (HAZ) through a domain. Rastering is achieved through the specification of layer-by-layer patterns, which can be combined into an overall pattern specifying the translation of the molten zone through the entire simulation domain. The application allows for arbitrary numbers of paths in each layer and an arbitrary number of layers in each pattern. Thus, the user can construct any scan strategy desired by specifying individual layer patterns and how these patterns should be repeated.

The molten pool is defined as a double ellipsoid. The ellipsoids share identical values for two of their axes (defined by the *melt_width* and *melt_depth* parameters). The third axis of each ellipsoid is defined by either the *melt_tail_length* or *cap_height* parameters. The haz is also defined by four equivalent parameters: *HAZ*, *tail_HAZ*, *depth_HAZ*, and *cap_HAZ*. A schematic of these eight parameters is shown below.



The model also requires specification of the *exp_factor* variable, which determines the value of the coefficient in the mobility equation, $M = exp(-exp\_factor * x)$, where x is the shortest distance from the lattice site to the molten pool boundary.

This application was used in the paper by Rodgers and collaborators.

The following additional commands are typically used by this application. A layer must be defined by using am cartesian_layer or am path_layer. A layer requires one to many am path commands or at least one am pass command.

- am pass: Specify pass parameters used to construct *cartesian_layer*.
- am path: Specify arbitrary paths via start/end points on a layer; sequence of *am path*s are used to construct an am path_layer.
- am cartesian_layer: A scan pattern on rectangular layer constructed from *am pass* and other parameters.
- am build: May be used for combinations of layers that comprise a pattern.

The examples/potts_additive directory has input files which illustrate use of these additional commands.

Restrictions:

This application is only compatible with square and square cubic lattices.

This application can only be evolved by a rejection KMC (rKMC) algorithm. See the sweep command for more details.

The settings for melt pool width + haz must be <= xhi & yhi.

Related commands:

app_style potts/weld, app_style potts/weld/jom

Default: none

(Rodgers) T.M. Rodgers, J.D. Madison and V. Tikare, "Simulation of Metal Additive Manufacturing Microstructures Using Kinetic Monte Carlo", Computational Materials Science (2017).

app_style chemistry command

Syntax:

```
app_style chemistry
```

- chemistry = application style name

Examples:

```
app_style chemistry
```

Description:

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

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

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

---

The following additional commands are defined by this application:

add_reaction	define a chemical reaction
add_species	define a chemical species
count	specify molecular count of a species
volume	specify volume of the chemical reactor

Restrictions: none

Related commands: none

Default: none

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

app_style diffusion command

Syntax:

```
app_style diffusion estyle dstyle args
```

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

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

Examples:

```
app_style diffusion linear hop
app_style diffusion nonlinear schwoebel 5 2
```

Description:

This is an on-lattice application which performs diffusive hops on a lattice whose sites are partially occupied and partially unoccupied (vacancies). It can be used to model surface diffusion or bulk diffusion on 2d or 3d lattices. It is equivalent to a 2-state Ising model performing Kawasaki dynamics where neighboring sites exchange their spins as the model evolves. Each lattice site stores a value which is 1 for vacant or 2 for occupied or 3 for vacant and a non-deposition site. See the deposition command for more details on the value = 3 sites.

Note that this application only allows for a single diffusing species (site value = 1) if run with an atomic-scale lattice, or a single phase if run with a coarse-grained lattice. See the app_style diffusion/multiphase command which allows for multiple diffusing species or phases.

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

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

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

```
Hi = Sum_j delta_ij
```

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

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

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

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

command. This effectively allows the energy to be a non-linear function of coordination number. As before the Hi for a vacant site is 0.

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

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

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

IMPORTANT NOTE: If you have a free surface you are depositing onto, it may also be possible for atoms to diffuse away from this surface, i.e. desorb into a vacuum. This application does not do anything special with those atoms (e.g. remove them), so they may clump together or induce deposition to take place onto the clumps above the surface. If you wish to prevent this you should insure that desorption is an energetically unfavorable event.

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

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

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

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

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

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

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

yes	no	up	finite	exp(-dE/kT)
yes	yes	down	0.0	0
yes	yes	down	finite	exp(-Q/kT)
yes	yes	up	0.0	0
yes	yes	up	finite	exp((-dE-Q)/kT)

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

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

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

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

barrier	define energy barriers for hop events
deposition	define deposition events
ecoord	specify site energy as a function of coordination number
temperature	set Monte Carlo temperature

Restrictions: none

Related commands:

app_style ising, app_style diffusion/multiphase

Default: none

app_style diffusion/multiphase command

Syntax:

```
app_style diffusion/multiphase
```

Examples:

```
app_style diffusion/multiphase
diffusion/multiphase pin 1           # after creating sites
diffusion/multiphase phase 2
diffusion/multiphase phase 3
diffusion/multiphase weight 0.5 pair 2 3
```

Description:

This is an on-lattice application which is a multi-species or multiphase extension to the single species or single phase diffusion app. If run with an atomic scale lattice, then this app allows definition of multiple atomic species via the diffusion/multiphase command it defines. Likewise if run with a coarse-grained lattice, it allows definition of multiple phases. The rest of this doc page uses the "phase" terminology.

In general, diffusion can lead to phase separation when bond energies (energy of a pair of neighboring sites) between like phases are lower than bond energies between different phases. The rate of phase separation and the degree to which it occurs can be controlled by setting the relative bond energies between different phases.

For this app, each site has a phase value which is a value >= 1. There can be an arbitrary number of phases. Note that if you want vacancies included in the model, they are specified as a distinct phase, just as Al or Cu atoms would be individual phases in an atomic-scale model.

As illustrated in the example above, the diffusion/multiphase command is used with this application. Every numeric phase (unique site value) must be set to either "phase" or "pin". The "phase" keyword should be used if the phase is mobile (it diffuses). The "pin" keyword should be used if it is immobile. The "weight" keyword is used to define a pairwise energy between pairs of neighbor phases in the energy Hamiltonian for the model:

```
Hi = Sum_j weight_ij
```

where Sum_j is a sum over all the neighbor sites of site I and weight_ij is the pairwise energy for the phases of site I and J. Only pairs of unlike phases can be assigned a weight. Pairs of like phases do not contribute to the site energy. See the diffusion/multiphase command for details.

Note that this equation means this app is effectively limited to the energy style *linear* option of the app_style diffusion command.

Also note that there should always be two or more non-pinned phases in your model. Otherwise no diffusive exchanges between sites with different phases will take place.

To run this application, an initial phase distribution on the lattice should be specified. Each lattice site has an integer value which stores its phase label. If only relative volume fractions of the phases is desired, the set command can be used to set values. If there is structure to the initial phase distribution, this can be written to a SPPARKS input file and read via the read_sites command; or, the set stitch command can be used to read phase distribution from a Stitch file formatted by the Stitch library included with SPPARKS.

The examples/diffusion_multiphase directory has several scripts which illusrate use of this app; example scripts also demonstrate initializing phase distributions.

- in.pin_filler
- in.pairwise_weighs
- in.variable_volume_fraction

The following additional command is defined by this application.

diffusion/multiphase	define phases and weights

Restrictions: none

Related commands:

diffusion/multiphase, app_style diffusion

Default: none

app_style erbium command

Syntax:

```
app_style erbium
```

- erbium = style name of this application

Examples:

```
app_style erbium
```

Description:

This application simulates a model of reaction and diffusion on a specialized Erbium lattice, which consists of an fcc lattice for the Erbium and additional tetrahedral and octahedral interstitial sites.

This application stores 2 integers per lattice site. The first integer (i1) is the "type" of the site:

- type = 1 = fcc
- type = 2 = octahedral
- type = 3 = tetrahedral

The second integer (i2) is the element on the site:

- element = 1 = erbium
- element = 2 = hydrogen
- element = 3 = helium
- element = 4 = vacancy

The 3-fold lattice should be created using the lattice fcc/octa/tetra command, which gives details of its geometry and neighbor connectivity.

The 3-fold lattice should normally be initialized in the following way, using the set command. All fcc sites are for erbium atoms and are fully occupied. All octahedral sites are initially vacant. A fraction of the tetrahedral sites is initialized with hydrogen atoms; the remainder are vacant.

The event command is used to define what kinds of diffusive hops and reaction events occur in the model. These can include correlated hops where a central site coordinates a change at two of its neighbor sites. Reaction events that transmute a Hydrogen atom to a Helium are also possible.

As explained on this page, this application can be evolved only by a kinetic Monte Carlo (KMC). You must thus define a KMC solver to be used with the application via the solve_style command.

For solution by a KMC algorithm, the list of events that can occur at each site is determined by its current neighbors and by the events specified via the event command. The relative probability of each event occurring is computed as a function of the rate or energy value specified in the event command and the temperature specified via the temperature command. The details are explained in the doc page for the event command.

The following additional commands are defined by this application.

event	definition of an event on the 3-fold lattice
temperature	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
```

- style = *ising* or *ising/single*

Examples:

```
app_style ising
app_style ising/single
```

Description:

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

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

```
Hi = Sum_j delta_ij
```

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

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

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

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

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

The following additional commands are defined by this application:

temperature	set Monte Carlo temperature

Restrictions: none

Related commands:

app_style potts

Default: none

app_style membrane command

Syntax:

```
app_style membrane w01 w11 mu
```

- 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)

Examples:

```
app_style membrane 1.25 1.0 -3.0
```

Description:

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

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

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

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

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

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

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

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

The following additional commands are defined by these applications:

inclusion	specify which sites are proteins
temperature	set Monte Carlo temperature

Restrictions: none

Related commands: none

Default: none

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

app_style phasefield/potts command

Syntax:

```
app_style phasefield/potts Q PFsteps lambda M_C KappaC a C1 C2 C3 C4 keyword value ...
```

- phasefield/potts = application style name
- Q = number of spin states
- PFsteps = number of phase field iterations per KMC/rMC iteration
- lambda = concetration free energy parameter
- M_C = Cahn-Hilliard mobility parameter
- KappaC = Cahn-Hilliard interfacial energy parameter
- a = bulk free energy parameter
- C1 = bulk free energy parameter
- C2 = bulk free energy parameter
- C3 = bulk free energy parameter
- C4 = bulk free energy parameter
- zero or more keyword/value pairs may be appended
- keyword = *reset_phasefield* or *print_connectivity* or *initialize_values* or *enforce_concentration_limits*

```
reset_phasefield value = yes or no
    if yes, enforce concentration boundary conditions: C(xlo) = 0, C(xhi) = 1.0
  print_connectivity value = yes or no
    if yes, print the neighborlist indices in the finite difference stencil
  initialize_values value = yes or no
    if yes, initializes phases with equilibrium concentration dE/dC = 0
  enforce_concentration_limits value = yes or no
    if yes, constrain site concentrations to the physical range of [0,1]
```

Examples:

```
app_style phasefield/potts 200 10 0.3 1 1 0.5 0.25 0.75 0.05 0.95
app_style phasefield/potts 200 10 0.3 1 1 0.5 0.25 0.75 0.05 0.95 &
          reset_phasefield yes &
          print_connectivity yes &
          initialize_values yes &
          enforce_concentration_limits yes
```

Description:

This is an on-lattice application which evolves a Q-state Potts model in combination with a phase field Cahn-Hilliard model. It can be used to efficiently simulate grain growth in a two-phase system controlled by diffusion. For a full description of the model, see the paper by Homer et al..

See the examples/potts_pfm directory for an example script using this command. See the Pictures web page for images of simulations performed with this command.

The following additional commands are defined by this application:

temperature	set Monte Carlo temperature

Restrictions: none

Related commands:

app_style potts

Default:

The keyword defaults are reset_phasefield = no, print_connectivity = no, initialize_values = no, enforce_concentration_limits = no.

(Homer) Homer, Tikare, Holm, Computational Materials Science, 69, 414-423 (2013).

app_style potts command

app_style potts/neigh command

app_style potts/neighonly command

Syntax:

```
app_style style Q
```

- style = *potts* or *potts/neigh* or *potts/neighonly*
- Q = number of spin states

Examples:

```
app_style potts 100
app_style potts/neigh 20
```

Description:

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

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

```
Hi = Sum_j delta_ij
```

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

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

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

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

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

1/4 probability. In all the cases, dE = Efinal - Einitial is calculated, as is a uniform random number R between 0 and 1. The new state is accepted if R < min[1,exp(-dE/kT)], else it is rejected.

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

The following additional commands are defined by these applications:

temperature	set Monte Carlo temperature

Restrictions: none

Related commands:

app_style ising

Default: none

app_style potts/am/bezier command

Syntax:

app_style potts/am/bezier nspins width depth h0 ht n

- potts/am/bezier = application style name
- nspins = number of possible spins
- width = maximum width of the melt pool
- depth = maximum depth of the melt pool
- h0 = heat affected zone distance at location of maximum pool width
- ht = heat affected zone distance at trailing edge of pool
- n = heat affected zone shape parameter; input values of n>=3/2

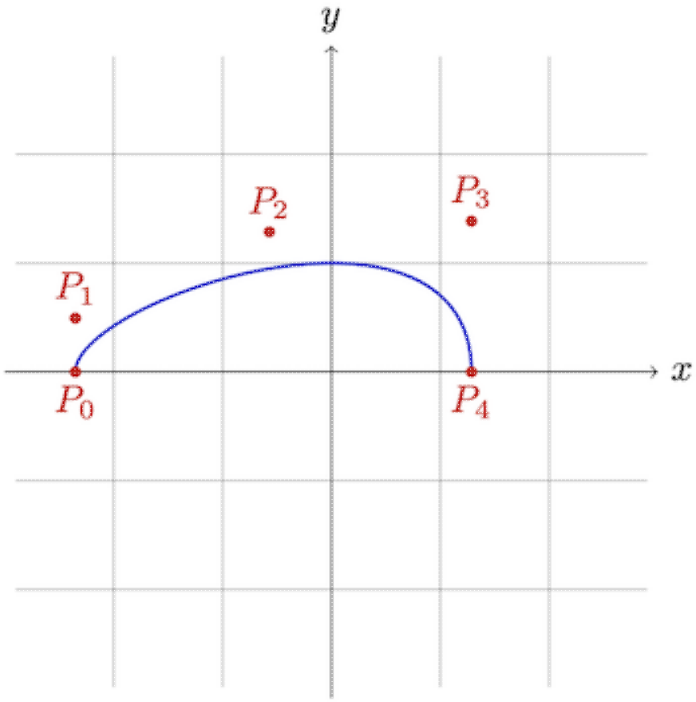
Examples:

app_style potts/am/bezier 250000 130 65 5

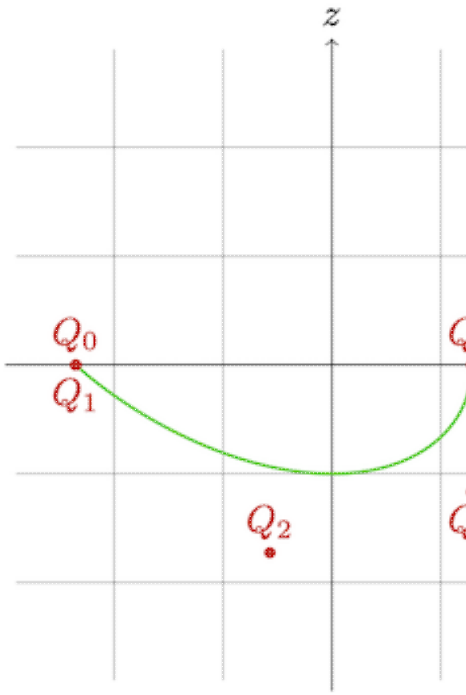
Description:

This is an on-lattice application derived from the app_style potts/neighonly application that simulates the rastering of a molten pool and its accompanying heat-affected zone (HAZ) through a domain. Rastering is achieved through the specification of layer-by-layer patterns, which can be combined into an overall pattern specifying the translation of the molten zone through the entire simulation domain. The application allows for arbitrary numbers of paths in each layer and an arbitrary number of layers in each pattern. Thus, the user can construct any scan strategy desired by specifying individual layer patterns and how these patterns should be repeated. Simulating arbitrary paths and layers is often used in conjunction with the Stitch IO via the set_stitch and dump stitch commands.

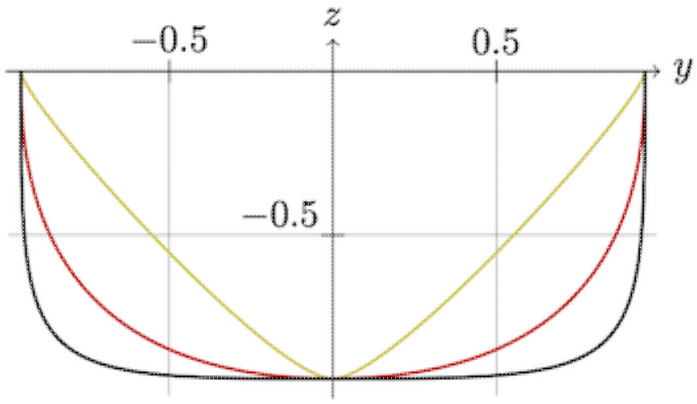
The app requires specification of melt pool dimensions *width,depth*, and parameters describing the surrounding heat affected zone *h0,ht,n*. The model also requires specification of the melt pool surface via the potts/am/bezier control_points command. Whereas app_am ellipsoid uses two ellipsoids to define the moving melt pool and heat effected zone, here, the molten pool is defined by two fourth order Bezier curves: 1) top surface curve; 2) spine curve. The two curves are combined to form a *3D* surface representing the interface between liquid and solidified material. The shape of the pool is defined by user input control points for the two curves and may be obtained from laboratory images or through process simulation or some other means.



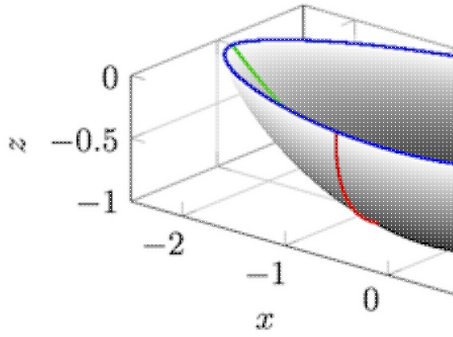
(a) Surface boundary with control points.



(b) Spine boundary with co



(c) Lateral cross section boundaries of the melt pool intersected with the plane $x = 0.5$ for $\langle \beta_y, \beta_z \rangle = \langle 0.2, 0.2 \rangle$, $\langle 1.0, 0.5 \rangle$, and $\langle 2.0, 1.0 \rangle$.



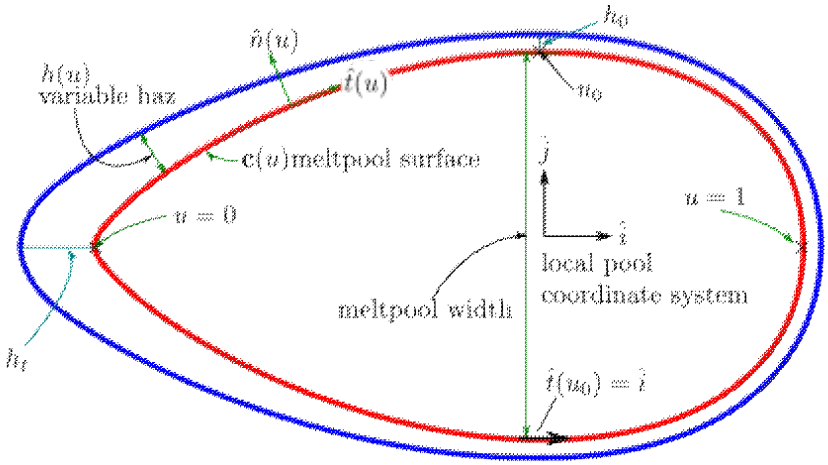
(d) Melt pool boundary with s lateral cross section boundary cu

Degree four Bezier curves require a total of five control points. However, symmetry of the melt pool across *xz-plane* and the bounding top surface plane *z=0* reduce the number of required inputs. Control points for the top surface curve are input only for half the melt pool on account of symmetry; because the front and tail of pool start at *y=0* only three *y* components of the top curve are required while the other two components are implicitly set to 0 by the app. Similarly, because the spine curve begins and ends at the top surface *z=0* only three components of *z* control points are required. Both the top surface curve and spine curve use the same *x* component values and all five are required inputs. Melt pool width and depth is explicitly set according to user inputs *width*, *depth* whereas melt pool length is implicitly defined by the ratio *width/length* according to the input top surface curve control points. The *x,y* components of the top curve control points are scaled to achieve the user input pool width. Similarly, spine curve control point components *x,z* are scaled to specified melt pool depth. However, the final set of *x* components used for both the top surface curve and spine curve are values obtained from the width scaling operation.

The following command defined by this application must be used to specify the bezier melt pool surface geometry as described above.

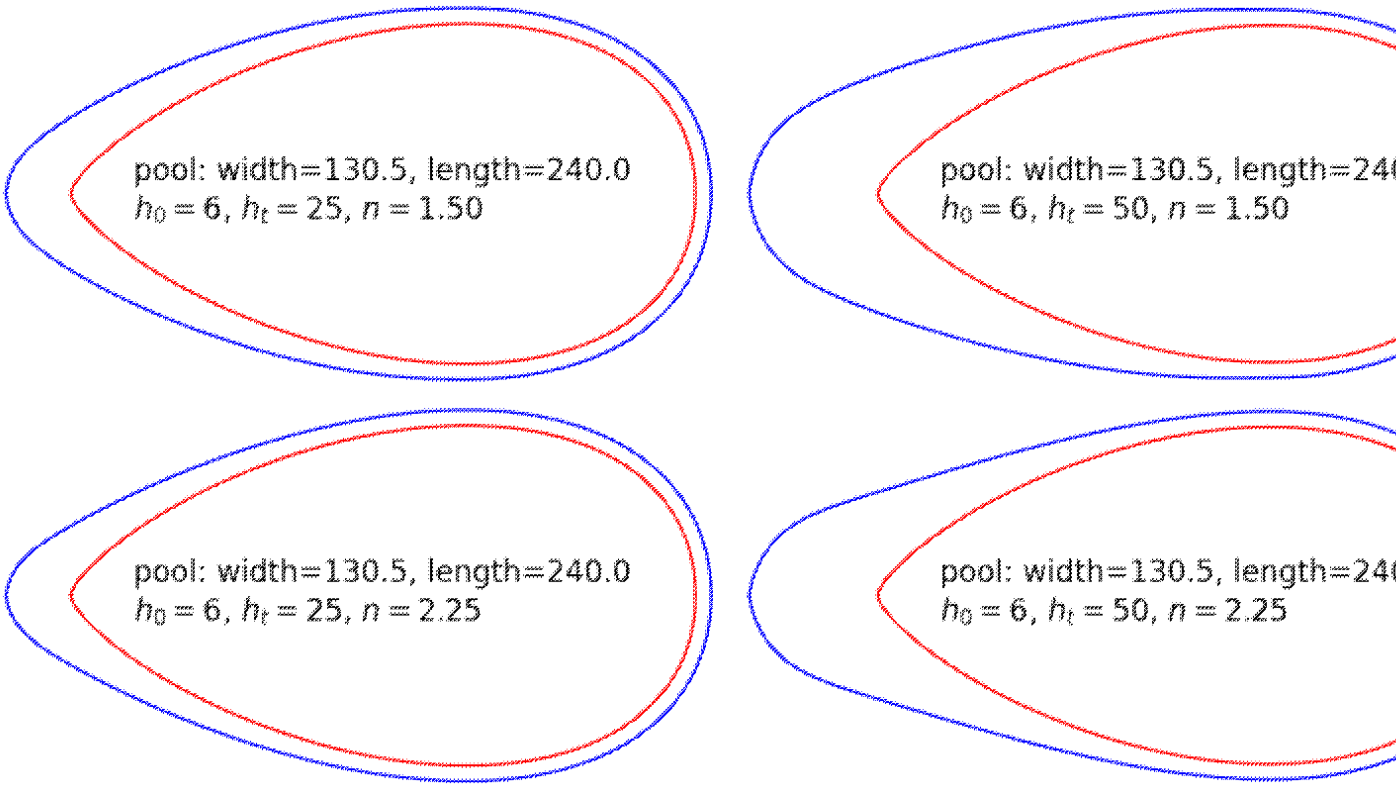
potts/am/bezier	Specifies control points and convexity of surface.

During the AM process, polycrystalline grains nucleate and evolve within the haz. This makes accurately capturing the haz shape important to microstructure prediction. In this model, three parameters *h0,ht,n* are used to describe the shape and extent of the haz. These parameters describe the variable heat affected zone that depends upon location relative to the pool surface, shown in red below. The limit of the *haz* is shown by the blue curve. *h0* and *ht* are length scale parameters and *n* is a dimensionless shape factor. Here, *h0* represents the size of the haz at the maximum pool width and depth, while *ht* indicates the length of the haz at the top surface trailing edge. Together, these parameters define a position-sensitive and variable haz.



Parameters *h0,ht* are input in spparks lattice units of length and characterize the heat-affected zone at the edge and trailing edge of the pool. The heat-affected zone is always present, which means that *h0* is always greater than zero. Additionally, *ht* should be greater than or equal to *h0*.

The dimensionless parameter *n* describes the haz shape. The value for *n* should be greater than *3/2*; a wide range of possible haz zone shapes are possible but care should be taken. A python script is given in the examples directory and was used to make the plots below for a fixed set of control points and varying set of haz parameters. With fixed control points, melt pool dimensions are the same. Different values for the variable haz parameters show changes in the resulting heat affected zone.



Some combination of the following additional commands are typically used by this application to specify a raster pattern. A layer must be defined by using am cartesian_layer or am path_layer. A layer requires one to many am path commands or at least one am pass command.

- am pass: Specify pass parameters used to construct *cartesian_layer*.
- am path: Specify arbitrary paths via start/end points on a layer; sequence of *am path*s are used to construct an am path_layer.
- am cartesian_layer: A scan pattern on rectangular layer constructed from *am pass* and other parameters.
- am build: May be used for combinations of layers that comprise a pattern.

The examples/potts_am_bezier directory has input files which illustrate use of some of these commands. However these commands are not unique to this app and other examples within the examples directory may also further illustrate.

Restrictions:

This application is only compatible with square and square cubic lattices.

This application can only be evolved by a rejection KMC (rKMC) algorithm. See the sweep command for more details.

Related commands:

app_style am/ellipsoid, app_style potts/weld, app_style potts/weld/jom

Default: none

Details of the melt pool representation used in this app are described in A BÃ©zier curve informed melt pool geometry for modeling additive manufacturing microstructures, Jeremy E Trageser, John A Mitchell, Kyle L Johnson, Theron M Rodgers, Computer Methods in Applied Mechanics and Engineering, Volume 415, 1 October 2023 116208.

app_style potts/am/path/gen command

Syntax:

```
app_style potts/am/path/gen
```

- potts/am/path/gen = application style name

Examples:

```
app_style potts/am/path/gen
```

This app does not have any app specific parameters. It does however require AM raster commands listed below.

Description:

This is a specialized additive manufacturing (AM) application. The app generates a sequence of computational volumes (CV) which correspond with AM raster paths on rectangular domains; it runs very quickly and does not need to be run in parallel. The sequence of CVs generated are ordered according to the am build specified; the union of CVs forms the final desired 3D rectangular volume over which microstructures are simulated.

Microstructure simulations on the sequence of CVs emulates the additive manufacturing process by appending incremental results to the output database much the same way material is added to a part during an AM build. SPPARKS simulations can be conducted on each CV and stitched together to form the final built part. AM microstructure simulations conducted this way require substantially reduced computational resources, both memory and mpi cpu resources, when compared to what would be required if the entire domain was instantiated for one large simulation. Starting and stopping the sequence of runs is also a highly flexible restart capability for AM microstructure simulations.

To generated CV and raster path information, the am pathgen command, supplied by this app, must be in the user input script; the script must also include am cartesian_layer and am pass commands which specify the rectangular domain as well as raster path information. The script should not use the create_sites commad -- more on this below.

The directory examples/stitch/stitching_rectangular_domain contains an example demonstrating use of this app. There are 3 primary elements to the example: 1) in.path_gen -- input script which uses this app; 2) in.am -- input script for a generic AM SPPARKS simulation on a CV parameterized; 3) stitch_rectangle.sh -- bash script which orchestrates the overall set of simulations. Very limited editing of bash script is required; only the spparks executable path is needed at top of script. Remaining parameters should be specified by in.path_gen and in.am. See directory for further explanation.

The following commands are needed and required by this application.

- region: Specify the larger intended simulation domain for the AM microstructure simulation. This domain will be decomposed by the app into a series of significantly small simulation domains (CVs).
- create_box: command to create region specified
- am pass: Specify pass parameters used to construct *cartesian_layer*.
- am cartesian_layer: A scan pattern on rectangular layer constructed from *am pass* and other parameters.
- am pathgen: May be used for combinations of layers that comprise a pattern. The command must be specified in the input script to generate output CV and raster information and must come after all of the above

commands.

Restrictions:

Do not use the create_sites command with this application. For this app to function properly, the region and create_box commands should be used to specify the larger intended simulation domain. If *create_sites* is used, the app will attempt to create a lattice on the larger intended simulation domain potentially requiring huge distributed memory requirements. If on the other hand the lattice is not created, the app is extremely light weight and can be used to generate the sequence of CVs and raster information on the larger intended domain. Thus the app can easily run in serial for any domain size provided the *create_sites* command is not used. It is not necessary to run the app in parallel although it will do so gracefully.

This application will only generate paths specified by the am cartesian_layer command and associated am pass commands.

Related commands:

None beyond those listed above.

Default: none

app_style potts/am/weld command

Syntax:

app_style style nspins alpha beta haz

- style = potts/am/weld
- nspins = number of possible Potts model spins
- alpha = controls relative size of melt pool shape at bottom compared to top
- beta = Bezier control point parameter that defines curvature of melt pool shape through thickness
- haz = width of the heat affected zone (haz) surrounding the melt pool

Examples:

```
app_style potts/am/weld 10000 0.5 0.75 50.0
weld_shape_ellipse 100.0 150.0
```

The *app_style potts/am/weld* command uses a subset of the command parameters used by *app_style potts/weld* -- additional explanation provided below. For an explanation of the above commands and parameters, see potts/weld example. This application requires melt pool geometry commands -- per example above and explanation below.

This application requires rastering commands -- per explanation given below.

The pulse command is disabled in this application.

Description:

This is an application for 2D additive manufacturing simulations and is an adaptation of potts/weld; it can be viewed as a potts/weld model with rastering commands as defined and used by potts_additive. The 2D limitation of this app derives from the full penetration weld assumption; *potts/weld*; does not have a melt pool bottom that naturally occurs in an additive manufacturing build the *z*-direction. Thickness of the lattice in the z-direction is taken as the plate thickness per description in the potts/weld; such simulations will produce meaningful 3D microstructures. However, it is generally more efficient to run 2D calculations with this app using only one plane of lattice sites.

Application *potts/am/weld* command values are all associated with potts/weld and have identical interpretations/meanings for *potts/am/weld*. Some values, e.g. *yp,velocity*, used in *potts/weld*, have been moved to raster commands.

The following commands must be used to specify pool geometry:

weld_shape_ellipse	specify elliptical pool shape parameters
weld_shape_teardrop	specify teardrop pool shape parameters

The following additional commands are typically used by this application. A layer must be defined: one of am cartesian_layer or am path_layer. A layer requires 1 to many am path or at least 1 am pass. Since this application only runs 1 layer, the *am build* command described below is optional.

am pass: Specify pass parameters used to construct *cartesian_layer*.

am path: Specify arbitrary paths via start/end points on a layer; sequence of *am path*s are used to construct an am path_layer.

am cartesian_layer: A scan pattern on rectangular layer constructed from *am pass* and other parameters.

am build: May be used to combinations of layers that comprise a pattern.

NOTE: Because *potts/am/weld* is intended for *2D* simulations, only the first layer, as specified in the *am build* command, is used; this application does not simulate multilayer builds. The examples/potts_additive directory has input files which illustrate how to use the rastering commands.

The temperature command should be used to set simulation Monte Carlo temperature.

Restrictions:

This application is only compatible with square and square cubic lattices. It can only be evolved by a rejection KMC (rKMC) algorithm. See sweep for more information.

Additional related commands:

app_style potts, app_style potts/grad, app_style am/ellipsoid, app_style potts/weld

Default: none

app_style potts/grad command

Syntax:

```
app_style style grad_style nspins m0 convert Q T0 grad_x grad_y grad_z
```

- style = *potts/grad*
- grad_style = *temp* or *mob*
- nspins = number of possible spins
- m0 = mobility constant for *temp* grad_style or mobility at the center of domain for *mob* grad_style
- convert = conversion factor for the gradients
- Q = activation energy
- T0 = temperature at the center of domain (temperature units)
- grad_x = gradient in the x direction
- grad_y = gradient in the y direction
- grad_z = gradient in the z direction

Examples:

```
app_style potts/grad temp 10000 .0006 .25 1 350 3 -3 1
app_style potts/grad temp 10000 .0006 .1  1 375 3.5 3
app_style potts/grad mob 15000  .5 1 0 0 .0024 0 0
```

Description:

This is an on-lattice application derived from the app_style potts/neighonly application which applies gradients given in three directions x,y, and z. The gradients can be either temperature or mobility gradients. If grad_style *mob* is chosen, mobility gradients are used. If grad_style *temp* is chosen, temperature gradients are used.

When the grad_style *temp* is used, the mobility of each site is assumed to depend on temperature, according to the equation m0 * exp(-Q/(KT)), where m0 is the mobility constant, K is Boltzmann's constant, T is the temperature of the site, and Q is the activation energy. The temperature of a site depends linearly on its position in the lattice. The linear function is uniquely defined by the value T0 at the center and the gradients in the x,y, and z directions, grad_x, grad_y, and grad_z, respectively. The gradients are in units of temperature per lattice spacing (defined by the lattice constant).

When the grad_style *mob* is used each site is assigned a mobility directly from the mobility gradients. The arguments Q and T0 are not used during a mobility gradient simulation. m0 is the initial mobility at the center of the domain. The mobility of each site depends linearly on its position in the lattice. The linear function is defined by the mobility gradients. The mobility gradients are in units of mobility per lattice spacing (defined by the lattice constant).

Under most circumstances a gradient will break periodicity in the gradient direction. This app requires a special method for turning off periodicity. Refer to the examples under *potts_grad* for more details on how to turn periodicity off.

Convert is an argument multiplied by the given gradients and is used to convert units as a convenience for the user.

Grad_z is an optional argument and will default to zero if not specified.

The following additional commands are defined by these applications:

temperature	set Monte Carlo temperature

Restrictions:

Convert must be > 0.

Only compatible with square and square cubic lattices.

Can only be evolved by a rejection KMC (rKMC) algorithm. See sweep for more information.

Nspins must be greater than the possible spins set with the set *site range* command.

Related commands:

app_style potts

Default:

As explained above the default for grad_z = 0.0.

(Garcia) A.L. Garcia, V. Tikare and E.A. Holm, "Three-Dimensional Simulation of Grain Growth of in a Thermal Gradient with Non-Uniform Grain Boundary Mobility", Scripta Met 59[6] 661- 664 (2008).

app_style potts/pin command

Syntax:

```
app_style potts/pin Q
```

- potts/pin = application style name
- Q = number of spin states

Examples:

```
app_style potts/pin 100
```

Description:

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

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

```
Hi = Sum_j delta_ij
```

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

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

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

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

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

The following additional commands are defined by this application:

pin	create a set of pinned sites
temperature	set Monte Carlo temperature

Restrictions: none

Related commands:

app_style potts

Default: none

app_style potts/quaternion command

Syntax:

```
app_style style Q crystal theta_cut
```

- style = *potts/quaternion*
- Q = number of spin states
- crystal = *cubic* or *hcp*
- theta_cut = optional positive cutoff angle (default=15.0) for Read-Shockley energy model

Examples:

```
app_style potts/quaternion 6400 cubic
app_style potts/quaternion 10000 hcp 25.0
```

Description:

This is an on-lattice application which evolves 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.

This Potts variant is designed to incorporate the effects of crystalline orientation into growth models, allowing more detailed and realistic simulations of microstructure evolution in Cubic and HCP polycrystals. Emulating crystalline orientation requires two additions to the standard Potts model: a choice of a reference crystal symmetry, and calculation of grain boundary energy based upon relative orientation differences between adjacent grains/sites.

The Hamiltonian representing the energy of site *i* is as follows:

```
Hi = (1/2) Sum_j e_ij
```

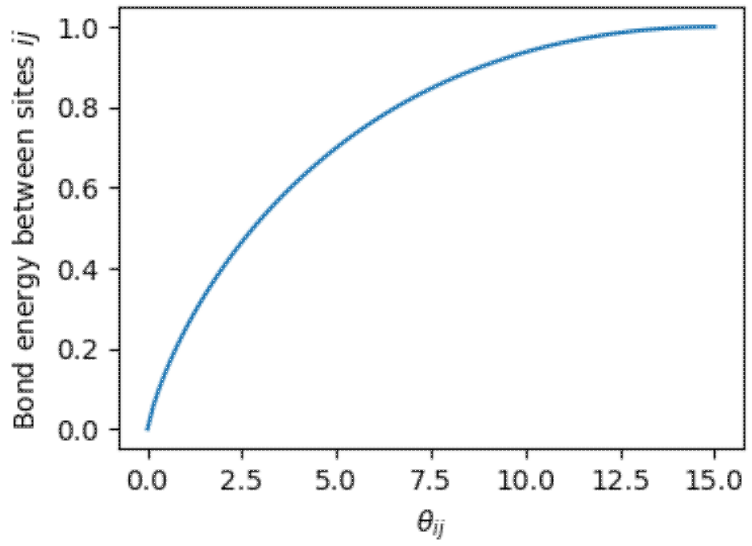
where Sum_j is a sum over all the neighbor sites *j* for site *i* and e_*ij* is the computed bond energy between sites *i* and *j*. Bond energy between sites *i* and *j* is computed using the Read-Shockley relation and the angular disorientation between sites *i* and *j*. The energy of the entire system is the sum of Hi over all sites.

The user selects either *cubic* or *hcp* crystal symmetry. To represent crystal orientation, each site *i* is initialized with a randomly generated unit quaternion, *qi*. The app calculates the disorientation angle *theta_ij* between two neighboring sites *i* and *j* using quaternions for the chosen crystal symmetry. The disorientation angle *theta_ij* is used to calculate grain boundary energy and evolve microstructure in exactly the same way the potts model is used to simulate grain growth and evolution without crystalline orientation.

More information on the calculation of *theta_ij* can be found in the Mackenzie and Handscomb papers (citations 1, 2, and 3 below). Python scripts in tools/potts_quaternion use the exact header files spparks uses; these python scripts can be used to verify disorientation distributions for random cubic and hcp orientations.

The default value for the low-angle cut-off *theta_cut* is 15.0 degrees, a commonly-used cutoff angle for cubic materials. To specify the low-angle cut-off different from the default, add the optional input value *theta_cut*. A maximum disorientation angle exists for each crystal symmetry: 62.7 degrees for *cubic* and 93.8 degrees for *hcp*. If the user's input *theta_cut* is greater than these maxima, the simulation will throw an error. A value *theta_cut=0* is disallowed and does not make sense for the Read-Shockley model.

The disorientation angle is used to calculate low-angle grain boundary energies using the Read-Shockley equation (citation 4). The plot below shows how grain boundary energy between sites *ij* varies with disorientation angle according to the Read-Shockley model.



Restrictions: none

Related commands:

app_style potts, app_style ising

Default:

theta_cut = 15.0

---

[1] Mackenzie, J. K., and M. J. Thomson. "Some statistics associated with the random disorientation of cubes." Biometrika 44, no. 1-2 (1957): 205-210. DOI: 10.2307/2333253

[2] Mackenzie, J. K. "Second paper on statistics associated with the random disorientation of cubes." Biometrika 45, no. 1-2 (1958): 229-240. DOI: 10.2307/2333059

[3] Handscomb, D.C. "On the Random Distribution of two Cubes", Canadian Journal of Mathematics , Volume 10 , 1958 , pp. 85 - 88 DOI: https://doi.org/10.4153/CJM-1958-010-0

[4] Read, W.T. and Shockley, W. "Dislocation models of crystal grain boundaries." Phys. Rev. 78, no. 3 (1950): 275. DOI: 10.1103/PhysRev.78.275

app_style potts/strain command

Syntax:

```
app_style potts/strain Q
```

- potts/strain = application style name
- Q = number of spin states

Examples:

```
app_style potts/strain 100
```

Description:

This is an on-lattice application which evolve a Q-state Potts model with a per-site strain, where each lattice site has a spin value from 1 to Q. Sites flip their spin as the model evolves. The strain energy can influence the grain growth.

The Hamiltonian representing the energy of site I is the same as for the Potts model:

```
Hi = Sum_j delta_ij
```

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

The per-site strain influences spin flips through altering the effective temperature as discussed below.

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

As explained on this page, this application is evolved by a kinetic Monte Carlo (KMC) algorithm. You must thus define a KMC solver to be used with the application via the solve_style command.

For solution by a KMC algorithm, a site event is a spin flip and its probability is 1/(1+strain) when dE <= 0 and exp(-dE/kT*) when dE > 0 and the temperature T is finite, where dE = Efinal - Einitial using the Hamiltonian defined above for the energy of the site, T is the temperature of the system defined by the temperature command (which includes the Boltzmann constant k implicitly), and T* = T (1 + strain). Thus the effect of the strain, defined for each site, is to rescale the temperature.

The KMC algorithm does not allow spin flips known as "wild" flips, even at finite temperature. These are flips to values that are not equal to any neighbor site value.

Strain values are stored for each site as a "double" value. This means they can be assigned to each site using the "d1" keyword with the set command, or read in via the read_sites command.

The application does not change the strain assigned to each site as the simulation progresses. But if SPPARKS is built and used as a library, as discussed in this section of the manual, the driver program can alter the per-site settings. The "couple" directory of the LAMMPS molecular dynamics package includes a sample coupled LAMMPS/SPPARKS application which uses LAMMPS to compute strain values at each site of a snapshot of

grain structure produced by this application running in SPPARKS. The strains are passed back to SPPARKS periodically by the driver application so that more Monte Carlo dynamics can be performed.

The following additional command is defined by this application:

temperature	set Monte Carlo temperature

Restrictions: none

Related commands:

app_style potts

Default: none

app_style potts/strain/pin command

Syntax:

```
app_style potts/strain/pin Q
```

- potts/strain/pin = application style name
- Q = number of spin states

Examples:

```
app_style potts/strain/pin 100
```

Description:

This is an on-lattice application which evolves a Q-state Potts model with a per-site strain, where each lattice site has a spin value from 1 to Q. The application also allows for pinning of sites using a special spin value of Q+1. Sites flip their spin as the model evolves but sites are not allowed to flip to a pinned value; sites with pinned values also do not flip. Strain energy can influence grain growth.

See the app_style potts_strain and pin commands for details on how strain and pinned sites are incorporated into this application.

The following additional commands are defined by this application:

temperature	set Monte Carlo temperature
pin	set pinning parameters

Restrictions: none

Related commands:

app_style potts, app_style potts_strain, pin

Default: none

app_style potts/weld command

Syntax:

app_style style nspins yp alpha beta velocity haz

- style = potts/weld
- nspins = number of possible Potts model spins
- yp = initial melt pool position along y-axis
- alpha = controls relative size of melt pool shape at bottom compared to top
- beta = Bezier control point parameter that defines curvature of melt pool shape through thickness
- velocity = velocity of melt pool motion (lattice sites per Monte Carlo step)
- haz = width of the heat affected zone (haz) surrounding the melt pool

Examples:

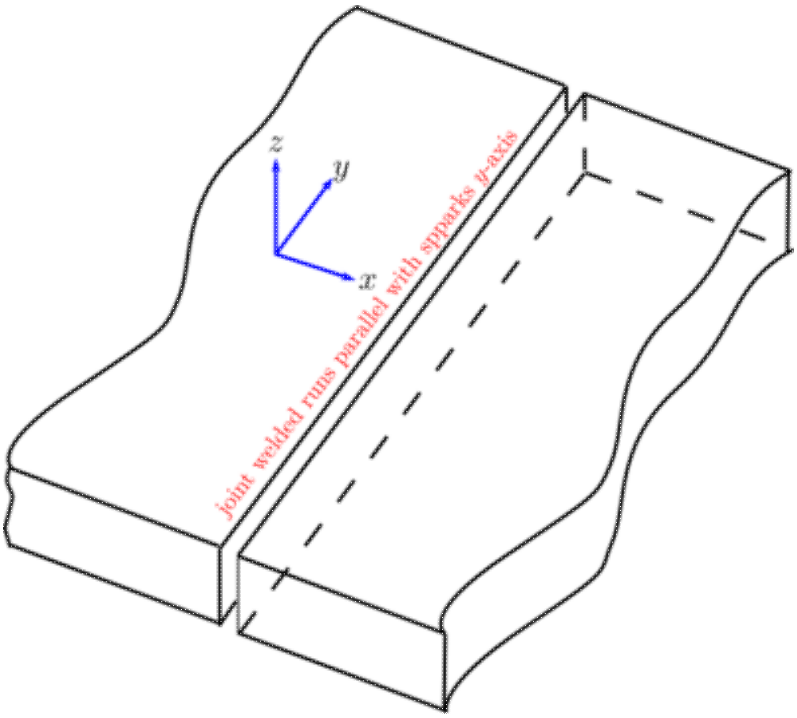
```
app_style potts/weld 10000 0 0.5 0.75 7.0 50.0
weld_shape_ellipse 100.0 150.0
```

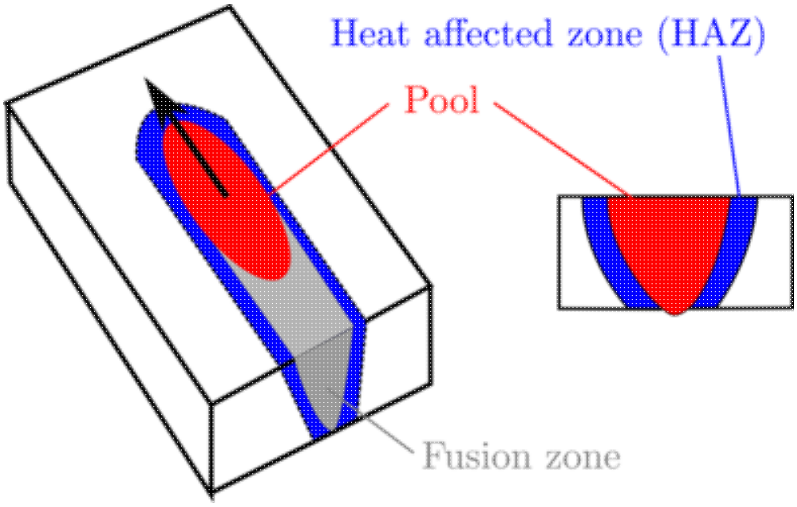
This defines a potts/weld model with 10000 spins. An elliptical pool shape is specified with width and length of 100 and 150 sites respectively; note these are pool dimensions at the top surface of the weld. The value *alpha=0.5* scales the elliptical pool width and length at the top surface to 50 and 75 sites respectively at the bottom (root) surface of the weld. The Bezier control point parameter specifies an outwardly curved pool; the weld speed is 7 MCS and the heat effect zone is 50 sites wide.

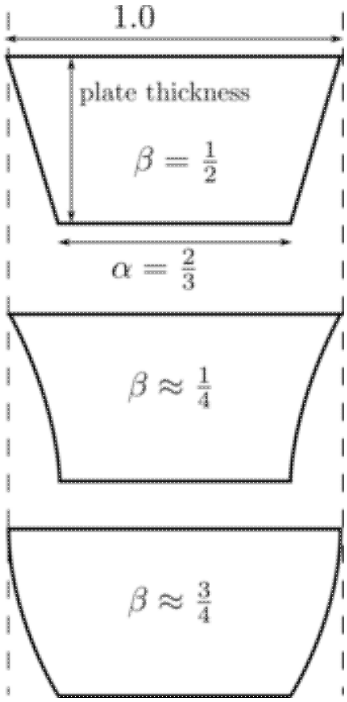
This application also requires one of the following commands to specify pool geometry:

weld_shape_ellipse	specify elliptical pool shape parameters
weld_shape_teardrop	specify teardrop pool shape parameters

Description:







This is an on-lattice application derived from the app_style potts/neighonly command. It simulates grain growth associated with a butt-weld process. Two sheets of material of equal thickness are assumed to be just touching; this defines a joint to be welded.

Grain growth associated with joining the plates in a weld process is simulated by translating a weld pool (simulates melt) along the joint (aligned with the y-coordinate axis). The weld pool is translated with a speed defined by *velocity*. The weld pool geometry is defined using an elliptical pool (weld_shape_ellipse) or a teardrop shaped pool (weld_shape_teardrop); these commands define the pool size and shape at the top surface of the plates joined. It is assumed that the weld fully penetrates the thickess of the plates; the parameter 1 >= *alpha* > 0 defines the the pool size at the bottom (root) of the plates relative to the top. The thickness of the plates joined is assumed to be the number of lattice sites along the z-axis defined in region command. Curvature of the pool in the plate thickness direction is controlled by the parameter 1 >= *beta* >= 0. When *beta*

is less than 0.5, the slope of the pool surface is increasing from top to bottom; when *beta* is greater than 0.5, the slope of the pool surface is decreasing from top to bottom.

The model simulates melting and re-solidification by randomizing the spin at a lattice site when it falls within the melt pool's volume. Upon exiting the melt pool, a rejection kinetic Monte Carlo event is performed at the site, and the spin is flipped to the value of one of its neighbors (in the style of the potts/neighonly application).

The mobility of each site within the *haz* region decreases linearly with increasing distance from the melt pool surface. The maximum mobility is 1 at the melt pool boundary and the minimum mobility is 0 at the outer boundary of the heat affected zone as defined by *haz*. The mobility gradient is similar to that in potts/grad, but is restricted to a smaller portion of the simulation domain as defined by the heat affected zone parameter *haz*.

Outside of the melt pool and heat affected zone, grain boundary mobility is set to 0, and grain evolution does not occur.

Use the read_sites command to initialize the microstructure of plates welded; alternatively the set command can be used to initialize the base metal microstructure.

The following additional commands are defined by this application:

weld_shape_ellipse	specify elliptical pool shape parameters
weld_shape_teardrop	specify teardrop pool shape parameters
pulse	apply pulsed arc power
temperature	set Monte Carlo temperature

Restrictions:

This application is only compatible with square and square cubic lattices. It can only be evolved by a rejection KMC (rKMC) algorithm. See sweep for more information.

Related commands:

app_style potts, app_style potts/grad

Default:

By default, this model runs without the affect of pulsed power.

app_style potts/weld/jom command

Syntax:

```
app_style potts/weld/jom nspins width length cap_length linear haz start_weld velocity weld_ty
```

- potts/weld/jom = application style name
- nspins = number of possible spins
- width = maximum width of the melt pool
- length = maximum length of the melt pool trailing the melt spot
- cap_length = specify the length of the heat source region leading the melt spot
- haz = width of the heat affected zone (haz) surrounding the melt pool
- start_weld = timestep at which to begin welding (usually 0)
- velocity = velocity of heat source motion (lattice sites per Monte Carlo step)
- weld_type = heat source shape (valid options are 1-5, see below)
- exp_factor = rate of exponential decay of the haz mobility gradient

Examples:

```
app_style potts/weld/jom 10000 30 25 10 40 0 10.0 1 0.01
```

Description:

This is an on-lattice application derived from the app_style potts/neighonly application which simulates a weld heat source traveling along the y-axis from y = 0 to yhi. The heat source is centered along the x-axis at x = xhi / 2 and top plane is located at zhi.

The model simulates melting and re-solidification by randomizing the spin at a lattice site when it falls within the melt pool's volume. Upon exiting the melt pool, a rejection kinetic Monte Carlo event is performed at the site, and the spin is flipped to the value of one of its neighbors (in the style of the potts/neighonly application).

The mobility of each site within the haz decreases exponentially with increasing distance from the melt pool surface. The maximum mobility is 1 at the melt pool boundary and the minimum mobility is 0 at the outer haz boundary. The mobility gradient is similar to that in potts/grad, but is restricted to a portion of the simulation domain defined by haz. Outside of the melt pool and haz, grain boundary mobility is set to 0, and grain evolution does not occur.

This program was used in the paper by Rodgers et al.

There are five different heat source shapes available defined by the integer (between 1 and 5) of "weld_type":

- 1 = "ellipsoid", An Goldak-style double ellipsoid heat source whose melt pool dimensions are defined with "width", "length", "cap_length", and ellipsoid_depth
- 2 = "keyhole", A keyhole heat source comprised of the union of two ellipsoids. A "shallow" ellipsoid whose dimensions are defined with "width", "length", and ellipsoid_depth, and a "deep" ellipsoid whose dimensions are defined with deep_width and deep_length. The "deep" ellipsoid is assumed to penetrate the entire depth of the simulation domain
- 3 = "linear", A heat source with linearly varying boundaries. The heat source's cross-section is constant along the z-axis

- 4 = "cap", A heat source with a power-law dependent boundaries. The heat source's cross-section is constant along the z-axis
- 5 = "circle", A heat source with circular boundaries. The heat source's cross-section is constant along the z-axis

---

The following additional commands are defined by this application:

ellipsoid_depth	define the maximum depth of the ellipsoid-shaped melt pool, or the maximum depth of the shallow melt pool in the keyhole model
deep_width	define the maximum width of the deep ellipsoid in the keyhole model
deep_length	define the maximum length of the deep ellipsoid in the keyhole model

---

Restrictions:

Only compatible with square and square cubic lattices.

Can only be evolved by a rejection KMC (rKMC) algorithm. See sweep for more information.

Melt pool width + haz must be =< xhi.

Related commands:

app_style potts, app_style potts/grad, app_style potts/weld

Default: none

(Rodgers) T.M. Rodgers, J.D. Madison and V. Tikare, "Predicting Mesoscale Microstructural Evolution in Electron Beam Welding", JOM 68[5] 1419- 1426 (2016).

app_style relax command

Syntax:

```
app_style relax delta
```

- relax = style name of this application
- delta = maximum displacement distance of a particle (distance units)

Examples:

```
app_style relax 0.5
```

Description:

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

The energy of a particle I is as follows:

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

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

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

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

The following additional commands are defined by this application:

temperature	set Monte Carlo temperature

Restrictions: none

Related commands:

Default: none

app_style sinter command

Syntax:

```
app_style style
```

style = *sinter*

Examples:

app_style sinter

Description:

This is on-lattice application which evolve a N-state Ising model or Potts model of sintering. Each lattice site has a spin value from -1 to N, with values of 0 representing internal pores sites, positive values representing grain sites and values of -1 representing the space outside the sintering compact. Sites change their spin to simulate microstructural evolution during sintering.

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

Hi = Sum_j delta_ij

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

A complete description of the sintering model and its parameters can be found in the 2010 Tikare, et.al. paper below.

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

As currently implemented, this application can be evolved only by the rejection KMC (rKMC) algorithm. You must thus define a sweeping method to be used with the application via the sweep command.

For solution by a rKMC algorithm, three different events are programmed: grain growth, pore migration and vacancy creation and annihilation. If the site selected is a grain site, a grain growth event is attempted: a new spin is chosen randomly from the values held by neighbor grain sites. If the site selected is a pore site, a pore migration or a vacancy creation and annihilation event is attempted. For the pore migration event, a new spin is chosen from the values held by neighboring grain sites such that the flip results in the minimum possible energy. For a vacancy creation and annihilation event, a pore site is moved to a neighboring grain site such that the resulting pore site is completely surrounded by grain sites (vacancy creation at grain boundary) with the subsequent movement of the vacancy to the surface of the sintering compact. In all the events, dE = Efinal - Einitial is calculated, as is a uniform random number R between 0 and 1. The new state is accepted if R < min[1,exp(-dE/kT)], else it is rejected. T is the temperature for simulating the event, so there is one temperature for grain growth, another for pore migration and a third temperature for vacancy creation and annihilation. These temperatures are defined by event_temperatures command (which includes the Boltzmann constant k implicitly).

Parallel implementation of the Monte Carlo model for sintering in SPPARKS code is described in the 2011 Garcia-Cardona paper below.

Initialization:

There are two methods to initialize the simulation space: randomly or from a file.

To do it randomly use the commands:

```
set i1 unique
```

- set i1 value 0 fraction p

The first command sets the spin of each site in the simulation space to a different value. The second command sets a fraction p of the spins to value 0, i.e. it defines the initial porosity of the sample. In this case, allow the grain structure to grow before starting sintering. This can be done by increasing the time to start the vacancy creation and annihilation event, using the time_sinter_start command.

To do it from a file:

See the documentation for the read_sites command. You should have defined previously the size of the corresponding simulation region and box by using the commands: region, create_box and create_sites

---

The following additional commands are defined by this application:

event_temperatures	set Monte Carlo temperature for each event
event_ratios	set frequency to attempt each event
time_sinter_start	set time to start attempting the vacancy creation and annihilation event

The following diagnostic styles are also useful with this application:

- sinter_avg_neck_area - calculate average neck area of the porous compact
- sinter_density - calculate density of the porous compact
- sinter_free_energy_pore - calculate surface pore area of the porous compact
- sinter_pore_curvature - calculate pore curvature of the porous compact

Use of the pore curvature to determine the sintering stress is described in the 2012 Garcia-Cardona paper below.

Restrictions: none

Related commands:

app_style potts

Default: none

(Tikare) V. Tikare, M. Braginsky, D. Bouvard and A. Vagnon, Numerical simulation of microstructural evolution during sintering at the mesoscale in a 3D powder compact, Comp. Mater. Sci., 48, 317-325 (2010).

(Garcia-Cardona) C. Garcia-Cardona, V. Tikare, S. J. Plimpton, Parallel simulation of 3D sintering, IJCMSSE,

4, 37-54 (2011).

(Garcia-Cardona2) C. Garcia-Cardona, V. Tikare, B. Patterson, E.A. Olevsky, On Sintering Stress in Complex Powder Compacts, J. Am. Ceram. Soc., 95, 2372-2382 (2012).

app_style sos command

Syntax:

```
app_style sos bond_energy keyword args
```

- sos = application style name
- bond_energy = lateral bond energy between columns
- zero or more keyword/value pairs may be appended
- keyword = *xsin*

```
xsin args = amp Lx Lz
    amp = amplitude of initial sine wave
    Lx  = wavelength of initial sine wave in x direction
    Lz  = wavelength pf initial sine wave in z direction (ignored if > 1.0e10)
```

Examples:

```
app_style sos 1.0 xsin 5.5 20.0 1.0e+20
app_style sos 2.0 none
```

Description:

The SOS (Solid-on-Solid) model is an on-lattice application that models a solid surface as a 1D or 2D lattice of sites. At each site an integer value represents the height of the surface at that site, so that collectively the heights of all the sites represent a surface profile with no overhangs or vacancies.

The Hamiltonian representing the energy of a site I is:

```
Hi = 1/2 J Sum_j |h_i - h_j|
```

where J is the bond energy, specified through the *bond_energy* parameter, and h_i and h_j are the heights at sites I and J. Sum_j represents a sum over the nearest neighbors of i, e.g. the neighbors to the immediate left and right for a 1D lattice.

If the *xsin* keyword is used, an initial height profile is assigned by a sine function. If the z dependence is inactive (Lz > 1.0e10), this is

```
hi = round(amp*sin(2*pi*x/Lx))
```

If the z dependence is active, this is instead:

```
hi = round(amp * min(sin(2*pi*x/Lx), sin(2*pi*z/Lz)))
```

This application performs Kawasaki dynamics, in which each event involves an "atom" hopping from one site to a neighboring site. That is, an event consists of site I losing one unit of height, and either site I+1 or I-1 simultaneously gaining one unit of height.

This application does not allow for use of a rejection KMC (rKMC) algorithm; only KMC options are supported. See this page for more information. For solution by a KMC algorithm, the probability of each "atom hop" event is min[P0, P0*exp(-dE/kT)], where P0 is a scaling factor, dE = Efinal - Einitial using the Hamiltonian defined above for the energy of the site, and T is the temperature of the system defined by the temperature command (which

includes the Boltzmann constant k implicitly). The scaling factor P0 is given by 1/nn where nn is the number of nearest neighbors for each site.

The following additional commands are defined by this application:

temperature	set Monte Carlo temperature

Restrictions: none

Related commands:

app_style diffusion

app_style command

Syntax:

```
app_style style args
```

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

Examples:

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

Description:

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

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

- diffusion = vacancy exchange diffusion model
- erbium = H/He diffusion/rection on an Erbium lattice
- ising = Ising model
- ising/single = variant Ising model
- membrane = membrane model of lipid,water,protein
- potts = Potts model for grain growth
- potts/neigh = variant Potts model
- potts/neighonly = variant Potts model
- potts/grad = Potts model with temperature gradient
- potts/pin = Potts model with pinning sites
- potts/strain = Potts model with per-site strain

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

- relax = Metropolis Monte Carlo relaxation

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

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

The general applications in SPPARKS can only be evolved via a kinetic Monte Carlo (KMC) solver, specified by the solve_style command. On-lattice and off-lattice applications can be evolved by either a KMC solver or a rejection kinetic Monte Carlo (rKMC) method or a Metropolis (MMC) method. The rKMC and MMC methods are specified by the sweep command. Not all on- and off-lattice applications support each option.

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

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

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

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

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

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

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

Restrictions: none

Related commands: none

Default: none

app_style test/group command

Syntax:

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

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

```
lomem value = yes or no
```

Examples:

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

Description:

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

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

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

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

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

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

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

```
run 10000
```

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

No additional commands are defined by this application.

Restrictions: none

Related commands:

solve_style group

Default:

The default value is lomem = no.

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

barrier command

Syntax:

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

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

Examples:

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

Description:

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

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

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

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

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

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

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

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

Restrictions:

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

Related commands:

deposition, ecoord

Default:

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

boundary command

Syntax:

```
dimension x y z
```

- x,y,z = *p* or *n* in each dimension

```
p is periodic
  n is non-periodic
```

Examples:

```
boundary p p n
```

Description:

Set the style of boundaries for the global simulation box in each dimension. The size of the simulation box is set by the create_box or read_sites commands.

The style *p* means the box is periodic in that dimension, so that sites can interact across the boundary.

The styles *n* means the box is non-periodic in that dimension, so that sties do not interact across the boundary.

Note that the interaction of a pair of neighboring sites is really controlled by each of their neighbor lists which are setup by either the create_sites or read sites commands. It is possible to have a periodic system with sites that do not interact across the periodic boundary, because of the way the neighbor lists of sites near the boundary are setup. See the create_sites or read sites for details.

IMPORTANT NOTE: The boundary command does not yet work with off-lattice applications.

Restrictions:

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

A 2d simulation must be periodic in the *z* dimesion. A 1d simulation must be periodic in the *y* and *z* dimensions.

Related commands:

dimension

Default:

```
boundary p p p
```

clear command

Syntax:

```
clear
```

Examples:

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

Description:

This command deletes all data, restores all settings to their default values, and frees all memory allocated by SPPARKS. Once a clear command has been executed, it is as if SPPARKS were starting over, with only the exceptions noted below. This command enables multiple jobs to be run sequentially from one input script.

These settings are not affected by a clear command: the working directory (shell command), log file status (log command), echo status (echo command), and input script variables (variable command).

Restrictions: none

Related commands: none

Default: none

count command

Syntax:

```
count species N
```

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

Examples:

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

Description:

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

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

Restrictions:

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

Related commands:

app_style chemistry, add_species, add_reaction

Default:

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

create_box command

Syntax:

```
create_box region-ID
```

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

Examples:

```
create_box mybox
```

Description:

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

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

Restrictions:

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

Related commands:

create_sites, region, read_sites

Default: none

create_sites command

Syntax:

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

- style = *box* or *region*

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

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

```
value values = label nvalue
    label = site or iN or dN
    nvalue = specific value to set all created sites to
  basis values = M nvalue
    M = which basis site (see asterisk form below)
    nvalue = specific value to set all created basis sites to
  global values = Xfull Yfull Zfull
    XYZ full = extent of global lattice to generate site IDs from
```

Examples:

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

boundary n n n
lattice sc/26n 1.0
create_box box block 101 200.1 1001 1200.1 301 400.1
create_sites box global 1000 2000 1000
```

Description:

This command creates "sites" on a lattice for on-lattice and off-lattice applications. For on-lattice applications it also defines a connectivity between sites that is stored as a neighbor list of nearby sites that each site interacts with.

This command is an alternative to reading in site coordinates and neighbor connectivity via the read_sites command.

To use this command, a simulation box must already exist, created via the create_box command. Likewise a lattice must also be defined using the lattice command.

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

This command generates the set of lattice points that fall within the simulation box. For any periodic dimension as specified by the boundary command, the origin of the lattice is the lower box boundary (in that dimension). Thus

coordinates begin at the lower boundary and increment by the lattice constant (in that dimension). The simulation box size must be an integer multiple of the lattice constant, to insure consistent placement of sites near periodic boundaries. SPPARKS is careful to put exactly one site on a periodic boundary (on the lower side of the box), not zero or two.

For non-periodic dimensions, the origin of the lattice is 0.0 (in that dimension). Thus coordinates begin at 0.0 and increment in both directions. Only coordinate inside the simulation box become sites. If a lattice point is inside or on a lower boundary (in that dimension), it is a site. Likewise if a lattice point is outside or on an upper boundary (in that dimension) it is considered outside the box. Thus for non-periodic dimensions you may need to tweak the simulation box size to get precisely the sites you want.

For the *box* style, all lattice points that fall inside the simulation box are stored as sites, as described in the preceding paragraphs. For the *region* style, a lattice point must additionally be consistent with the region volume to be stored as a site. Note that a region can be specified so that its volume is inside or outside a box boundary.

For on-lattice applications, after sites have been created, a neighbor list is also generated for each site, as defined by each lattice style. Think of this as the set of lattice points near a central site, with which it interacts in the sense defined by an application. If the simulation box is periodic in a dimension, the neighbors of a central site may include sites on the other side of the box. This will not be the case for a non-periodic dimension. If some sites do not exist, e.g. when using the *region* style, then those sites will not have a complete set of neighbors.

SPPARKS attempts to create sites with consecutive IDs from 1 to N, where N is the total number of sites that fill the simulation box. But it cannot always do this. In these scenarios consecutive IDs should be produced:

- *style* = box and the simulation box is fully periodic
- style = box, the simulation box is fully periodic or non-periodic (in one or more dimensions), a simple regular lattice is used, namely line (line/2n) for 1d models, square (sq/4n or sq/8n) for 2d, or simple cubic (sc/6n or sc/26n) for 3d, and the *global* keyword is not used

In the 2nd scenario the site IDs will vary fastest in x, then in y, and slowest in z. So it easy to use another program to generate values on a regular lattice associated with the correct IDs.

In all other cases, the site IDs may not be consecutive (1 to N). In particular, they may not be consecutive in any of these cases:

- *style* = region
- the *global* keyword is used
- the simulation box is non-periodic (in one or more dimensions) and the lattice is not one of the simple regular lattices listed above for 1d, 2d, or 3d

Regardless of what the site IDs are, they will be the same independent of the number of processors used to run the simulation.

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

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

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

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

The *global* keyword only affects generation of site IDs. It can only be used for on-lattice applications, for *style* = box, and for simple regular lattices. The latter requirement means lattice = line (line/2n) for 1d models, square (sq/4n or sq/8n) for 2d, or simple cubic (sc/6n or sc/26n) for 3d.

It is useful when a series of SPPARKS simulations are being run on a global lattice of sites that is larger than the simulation box for an individual simulation, e.g. in an additive manufacturing model. In this scenario, the per-site values used to initialize a simulation are typically read from a file (see the read_sites or set file commands) and the per-site values generated by the simulation are archived to a file (see the dump hdf5 command). In both cases the archive file contains sites for the entire global lattice and is accessed by site IDs. This command allows an individual SPPARKS simulation to generate site IDs that match those in the file for the global lattice.

The *Xfull*, *Yfull*, *Zfull* values are the size of the global lattice. Its site IDs are assumed to run from 1 to N = Xfull*Yfull*Zfull. Note that for a 2d model, Zfull = 1 is required. As described above for SPPARKS site IDs on a regular lattice, the global IDs vary fastest in x, then y, and slowest in z.

To use the *global* option correctly, the simulation box created by the create_box command must be specified appropriately.

If a dimension of the global lattice is intended to be non-periodic, because a single SPPARKS simulation will only model a portion of that dimension, then SPPARKS must set it to be non-periodic via the boundary command. And the lo/hi box boundaries in that dimension, as specified by the create_box command, should be set so that lattice sites are generated that correspond to the desired portion of the global lattice.

For example, imagine a global lattice that is 1000x2000 for a 2d simulation with both dimensions non-periodic. And you wish SPPARKS to model the lower left 100x100 corner of that global lattice. Assume the x and y lattice spacings are 1.0.

The following commands would setup the sites for this simulation:

```
dimension       2
boundary        p p p
lattice         sq/4n 1.0
region          box block 1 100.1 1 100.1 -0.5 0.5
create_box      box
create_sites    box global 1000 2000 1
```

Picture the global lattice as a 1000x2000 array of sites numbered with IDs ranging from 1 to 2 million, where the lower left corner has ID = 1, and the IDs increase fastest in x, and slowest in y. SPPARKS will create sites with coordinates and IDs corresponding to the lower left 100x100 corner of that array. I.e. the sites in the 100x100 SPPARKS model will be ordered as follows:

```
1,2,3, ..., 100                 # first row of x sites
1001,1002,1003, ... 1100        # next row of x sites
```

```
...
99001,99002,99003, ... 99100    # last row of x sites
```

Note the need to use *xhi* = *yhi* = 100.1, instead of 100.0, in the "create_box" command for the upper bound of non-periodic dimensions. This is because, as explained above, a non-periodic box will not generate sites that lie exactly on the upper-boundary (in any dimension). So if 100.0 were used, the size of the SPPARKS domain in that dimension would be one less than desired.

Similarly, the same commands with this substitution:

```
region           box block 901 1000.1 901 1000.1 -0.5 0.5
```

would model the upper right corner of the global lattice. The site at the lower left corner of the 100x100 SPPARKS simulation would have ID = 1900901; the upper-right corner site would have ID = 2 million.

Finally, if a dimension of the global lattice is intended to be periodic, then SPPARKS must set it to be periodic via the boundary command and each SPPARKS simulation must span that entire dimension. As described above, the simulation box size in that dimension must thus be N lattice units in size, where N = Nfull for that dimension. For example, if y is a periodic dimension, then the ylo and yhi parameters in the create_box command must be such that yhi-ylo = Yfull. Any pair of ylo,yhi values that satisfy this constraint can be used.

Here are more examples of several sets of SPPARKS create_sites commands using the *global* keyword, for 2d global lattices of 2 different sizes, with either periodic or non-periodic boundaries.

```
# global = 10x10, periodic in both x and y
# SPPARKS models 100 sites, must model entire global lattice
dimension        2
boundary         p p p
lattice          sq/4n 1.0
region           box block 0 10 0 10 -0.5 0.5
create_box       box
create_sites     box global 10 10 1

# global = 10x10, non-periodic in both dims
# SPPARKS models 25 sites = upper-left quarter
dimension        2
boundary         n n p
lattice          sq/4n 1.0
region           box block 1 5.1 6 10.1 -0.5 0.5
create_box       box
create_sites     box global 10 10 1

# global = 10x10, non-periodic in x, periodic in y
# SPPARKS models 50 sites = right half, must model entire y dim
dimension        2
boundary         n p p
lattice          sq/4n 1.0
region           box block 6 10.1 0 10 -0.5 0.5
create_box       box
create_sites     box global 10 10 1

# global = 10x10, periodic in x, non-periodic in y
# SPPARKS models 50 sites = middle section, must model entire x dim
dimension        2
boundary         p n p
lattice          sq/4n 1.0
region           box block 0 10 3 7.1 -0.5 0.5
create_box       box
create_sites     box global 10 10 1
```

```
# global = 100x100, non-periodic in both dims
# SPPARKS models 100 sites = upper left corner
dimension        2
boundary         n n p
lattice          sq/4n 1.0
region           box block 1 10.1 91 100.1 -0.5 0.5
create_box       box
create_sites     box global 100 100 1

# global = 100x100, non-periodic in both dims
# SPPARKS models 100 sites = lower middle section
dimension        2
boundary         n n p
lattice          sq/4n 1.0
region           box block 45.0 54.1 1 10.1 -0.5 0.5
create_box       box
create_sites     box global 100 100 1
```

Restrictions:

The app_style command must be used to define an application before using the create_sites command. The create_box command must be used to to define the simulation box before using the create_sites_command.

As explained above, the *global* keyword only affects generation of site IDs. It can only be used for on-lattice applications, for *style* = box, and for simple regular lattices. The latter requirement means lattice = line (line/2n) for 1d models, square (sq/4n or sq/8n) for 2d, or simple cubic (sc/6n or sc/26n) for 3d.

Related commands:

lattice, region, create_box, read_sites

Default: none

deep_length command

Syntax:

```
deep_length L
```

- L = Maximum length of the deep melt pool ellipsoid used in the keyhole weld model

Examples:

```
deep_length 20
```

Description:

This command is used in the keyhole mode of the potts/weld_jom application to define the maximum length of the deep ellipsoid, which extends through the entire z-axis of the domain.

Restrictions:

This command can only be used as part of the app_style potts/weld_jom application.

It must be a positive value.

Related commands:

deep_width, ellipsoid_depth

Default: 1/4 * yhi

deep_width command

Syntax:

```
deep_width W
```

- W = Maximum width of the deep ellipsoid used in the keyhole weld model

Examples:

```
deep_width 30
```

Description:

This command is used in the keyhole mode of the potts/weld_jom application to define the maximum width of the deep ellipsoid, which extends through the entire z-axis of the domain.

Restrictions:

This command can only be used as part of the app_style potts/weld_jom application.

It must be a positive value.

Related commands:

deep_length, ellipsoid_depth

Default: 1/3 * xhi

deposition command

Syntax:

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

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

Examples:

```
deposition       event 1.0 0 -1 0 1.0 1 4
deposition       batch 1.0 1 1 -1 1.0 3 10
deposition       off
```

Description:

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

If *mode* is set to *off*, then no additional arguments are used. Deposition is turned off. This can be useful when deposition previously took place, but is now turned off.

If *mode* is set to *event*, then deposition events will be performed in tandem with diffusive hop events in the KMC diffusion model. This option only works when running on a single processor.

If *mode* is set to *batch*, then deposition events will be performed as a batch at the end of each KMC loop over sectors. Thus diffusive events and deposition events are separated. This option only works when running in parallel on multiple processors.

For each trial deposition, a random starting point at the top of the simulation box is selected (top y surface in 2d, top z surface in 3d). The atom trajectory (straight line) is traced along its incident direction which is specified by (dirx,diry,dirz) and need not be a unit vector. However, diry < 0 and dirz = 0 is required for 2d models. Similarly, dirz < 0 is required for 3d models.

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

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

For the set of candidate sites, the selected deposition site is the one closest to the starting point, measuring the distance from the projected perpendicular point to the starting point.

IMPORTANT NOTE: App_style diffusion defines valid sites as vacant (site value = 1) or occupied (value = 2). When performing deposition, a row (2d) or plane (3d) of sites at the top of the system (where the deposited atoms are incident from) should be set to a value of 3. This prevents those sites from being considered as candidate deposition sites, due to them being neighbors of occupied sites at the bottom of the system in a periodic sense.

Restrictions:

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

Related commands:

ecoord, barrier

Default:

The default is mode = off.

diag_style array command

Syntax:

```
diag_style array value mode value mode ...
```

- array = style name of this diagnostic
- value = *iN* or *dN*
- mode = *min* or *max* or *mean* or *sum*

Examples:

```
diag_style array i2 mean
diag_style array d1 sum d1 min d1 max
```

Description:

The array diagnostic computes the mean, sum, min, or max for a per-site lattice value in the system. The diagnostic can operate on one or more values in one or more modes (min, max, mean, sum). The results are printed as stats output via the stats command.

Restrictions:

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

Related commands:

diag_style, stats

Default: none

diag_style cluster command

Syntax:

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

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

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

Examples:

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

Clustering can only be used with the lattice application, and applications based on it.

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 = *total number of sites*
- nprocs = *number of processors*

- Time = *time*
- ncluster = *total number of clusters*
- id ivalue dvalue size cx cy cz xlo xhi ylo yhi zlo zhi
- *cluster id ivalue dvalue size cx cy cz xlo xhi ylo yhi zlo zhi*

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. For most lattice

applications it is zero. *size* is the numbers of sites in the cluster.

Cx, *cy*, *cz* are the coordinates of the centroid of the cluster i.e. the average of the x, y, and z coordinate of all the sites in the cluster. For clusters than are of finite extent in a periodic dimension, the average is over the contiguous sites in a single periodic image, and the centroid is shifted by multiples of the period so as to lie inside the box. For clusters of infinite extent in x, y, or z, the centroid is not defined, so the clustering algorithm will produce a result based on some arbitrary splitting of the cluster into finite periodic repeat units. Except for this last case, the calculated *cx*, *cy*, or *cz* will be not be affected by the numbers of processors used in the calculation.

Xlo, *xhi*, *ylo*, *yhi*, *zlo*, and *zhi* are the maximum and minimum x, y, and z coordinates of sites in cluster, in other words the extent of the bounding box of the cluster. For clusters that are of finite extent in a periodic dimension, the max and min are taken over the contiguous sites in a single periodic image, and each of the 6 output values are then shifted by multiples of the period so as to lie inside the box. For clusters of infinite extent in x, y, or z, the max and min values in those directions are not defined. The clustering algorithm will produce a result based on some arbitrary splitting of the cluster into finite periodic repeat units. Except for this last case, the max and min values will be not be affected by the numbers of processors used in the calculation.

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

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

Restrictions:

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

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

Related commands:

diag_style, stats

Default: none

diag_style diffusion command

Syntax:

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

- diffusion = style name of this diagnostic zero or more keyword/value pairs may be appended
  - see the diag_style command for keyword/value pairs that can be appended to a diagnostic command

Examples:

```
diag_style diffusion
```

Description:

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

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

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

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

Restrictions:

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

Related commands:

diag_style, stats

Default: none

diag_style energy command

Syntax:

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

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

Examples:

```
diag_style energy
```

Description:

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

Restrictions:

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

Related commands:

diag_style, stats

Default: none

diag_style 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 keyword/value pairs 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 exist
    events = total # of events for all sites
    sN,dN,tN = cummulative # 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.

Following the *list* keyword you can list one or more of the listed values, in any order.

The *er*, *h*, *he*, and *vac* values will print counts of the number of current sites of each type. The *events* value will print the total # of possible events that can occur as defined by the event command, given the current state of the lattice, summed over all sites.

The *sN*, *dN*, and *tN* values refer to a tally of events that have actually occurred, as 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 values printed for *sN*, *dN*, and *tN* are cummulative counts of events 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 propensity command

Syntax:

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

- propensity = style name of this diagnostic zero or more keyword/value pairs may be appended
  - see the diag_style command for keyword/value pairs that can be appended to a diagnostic command

Examples:

```
diag_style propensity
```

Description:

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

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

Restrictions:

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

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

Related commands:

diag_style, stats

Default: none

diag_style sinter_avg_neck_area command

Syntax:

```
diag_style sinter_avg_neck_area keyword value keyword value ...
```

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

Examples:

```
diag_style sinter_avg_neck_area
```

Description:

The sinter average neck area diagnostic computes the average neck area in the powder compact simulated. The average neck area is printed as stats output via the stats command.

Restrictions:

This diagnostic can only be used for the sintering application.

Related commands:

diag_style, stats

Default: none

diag_style sinter_density command

Syntax:

```
diag_style sinter_density keyword value keyword value ...
```

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

Examples:

```
diag_style sinter_density
```

Description:

The sinter density diagnostic computes the density of the powder compact simulated. The calculation is done over the 1/27th central parallelepiped in order to avoid border effects. The density is printed as stats output via the stats command.

Restrictions:

This diagnostic can only be used for the sintering application.

Related commands:

diag_style, stats

Default: none

diag_style sinter_free_energy_pore command

Syntax:

```
diag_style sinter_free_energy_pore keyword value keyword value ...
```

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

Examples:

```
diag_style sinter_free_energy_pore
```

Description:

The sinter free energy pore diagnostic computes the surface area of the pores in the powder compact simulated. The calculation is done over the 1/27th central parallelepiped in order to avoid border effects. To obtain a measure independent of the size of the simulation the value computed is normalized by dividing over the volume used. The pore free energy is printed as stats output via the The density is printed as stats output via the stats command.

Restrictions:

This diagnostic can only be used for the sintering application.

Related commands:

diag_style, stats

Default: none

diag_style sinter_pore_curvature command

Syntax:

```
diag_style sinter_pore_curvature keyword value keyword value ...
```

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

Examples:

```
diag_style sinter_pore_curvature
```

Description:

The sinter pore curvature diagnostic computes the mean integral curvature of the pores in the powder compact simulated. In addition the triple line length is also computed. The calculation is done over the 1/27th central parallelepiped in order to avoid border effects. The pore curvature and the triple line length are printed as stats output via the stats command.

The method used to measure pore curvature is described in detail in "On Sintering Stress in Complex Powder Compacts", Cristina G. Cardona, Veena Tikare, Burton R. Patterson and Eugene Olevsky, J. Am. Ceram. Soc., 1-11 (2012)

Restrictions:

This diagnostic can only be used for the sintering application.

Related commands:

diag_style, stats

Default: none

diag_style command

Syntax:

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

- style = *cluster* or *diffusion* or *energy* or *propensity*
- zero or more keyword/value pairs may be appended
- keyword = *stats* or *delay* or *delt* or *logfreq* or *loglinfreq*

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

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

Examples:

```
diag_style cluster stats no delt 1.0
diag_style energy
```

Description:

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

The diagnostics currently available are:

- array = statistics of lattice values
- cluster = grain size statistics for general lattices
- diffusion = statistics on diffusion events
- energy = energy of entire system for general lattices
- propensity = propensity of entire system for general lattices

---

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

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

If *stats* is set to *no*, then the other keywords related to how often the diagnostic is called may be used. The *delt* keyword specifies *Delta* = the interval of time between each diagnostic calculation.

Similarly, the *logfreq* and *loglinfreq* keywords will cause the diagnostic to run at progressively larger 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. See the stats command for more information on how the output times are specified. See the stats command for more information on how the intervals are specified.

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

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

Restrictions: none

Related commands:

stats

Default:

The stats setting is yes.

diffusion/multiphase command

Syntax:

```
diffusion/multiphase keyword args
```

- keyword = *phase* or *pin* or *weight*

```
phase arg = ID
    ID = phase label (integer)
  pin arg = ID
    ID = phase label (integer) for a pinned phase
  weight args = w pair i j
    w = weight applied between phases i and j
    pair = required keyword
    i,j = pair of phase labels
```

Examples:

```
diffusion/multiphase pin 1
diffusion/multiphase phase 2
diffusion/multiphase phase 3
diffusion/multiphase weight 0.5 pair 2 3
```

Description:

This command is used with the app_style diffusion/multiphase application.

The command is typically used multiple times, each time with one of 3 keywords.

The command must be used once for each defined phase, either with the *phase* keyword or the *pin* keyword.

The *phase* keyword defines a mobile phase (which can diffuse) with an integer phase label.

The *pin* keyword defines an immobile phase with an integer phase label. Pinned sites never exchange values with another site, i.e. the sites in a pinned phase do not diffuse.

There is no limit to the number of phases which can be defined. However there should always be two or more non-pinned phases in your model. Otherwise no diffusive exchanges between sites with different phases will take place.

The *weight* keyword specifies a pairwise bond energy (weight) between two neighboring sites with the specified I,J phase values. The default weight for all pairs of unlike phases is 1.0. A weight specified for an I,J pair will also be applied to J,I. Note that a weight cannot be assigned to an I,I pair; a pair of I,I neighbors do not contribute to the energy of either site.

In the 4 example lines above, 3 phases are defined. Phase 1 is pinned, phases 2 and 3 are not. A weight of 0.5 is applied to the 2,3 pair of phases. The weights for the 1,2 and 1,3 pairing are the default values of 1.0.

Restrictions:

This command can only be used with the app_style diffusion/multiphase application.

Related commands:

app_style diffusion/multiphase

Default:

When pairwise weights are not defined, weights values default to 1.0.

dimension command

Syntax:

```
dimension N
```

- N = 1 or 2 or 3

Examples:

```
dimension 2
```

Description:

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

Restrictions:

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

Related commands: none

Default:

```
dimension 3
```

dump command

dump image command

Syntax:

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

- dump-ID = user-assigned name for the dump
- style = *text* or *sites* or *vtk* or *stitch* or *image*
- delta = time increment between dumps (seconds)
- filename = name of file to dump snapshots to
- fields = list of arguments for a particular style

```
text or sites or stitch or vtk fields =
     id or site or x or y or z or
     energy or propensity or iN or dN
  image fields = discussed on dump image doc page
```

Examples:

```
dump 1 text 0.25 tmp.dump
dump 1 text 1.0 my.dump id site x y z
dump 1 sites 1.0 my.sites.* id site i2 i3
dump 1 vtk 1.0 my.vkt.* site
dump 1 stitch 1.0 stitch_file.st site
dump mydump text 5.0 snap.ising id site energy i1
```

Description:

The *text*, *sites*, *vtk*, and *stitch* styles dump a snapshot of site values to one or more files at time intervals of *delta* during a simulation. The *image* style creates a JPG or PPM image file of the site configuration every at time intervals of *delta*, as discussed on the dump image doc page. The remainder of this page describes the *text*, *sites*, *vtk*, and *stitch* styles.

The *text* style dump file is in the format of a LAMMPS dump file which can thus be read-in by the Pizza.py toolkit, converted to other formats, or used for visualization.

The *sites* style dump file is in the same format that is read by the read_sites command. The dumped files can thus be used as restart files to continue a simulation, using the read_sites command.

The *vkt* style dump file is in the VTI format that can be read by various visualization programs, including ParaView.

The *stitch* style dump file is in an SQLite format which can be read by the set stitch command or auxiliary tools provided with the Stitch library in lib/stitch. See the examples/stitch dir for examples of SPPARKS scripts that read and write *stitch* files.

As described below, the filename determines the kind of output (text or binary or gzipped, one big file or one per timestep, one big file or one per processor or one file per group of processors). The fields included in each snapshot are obtained from the application. Only on-lattice and off-lattice applications support dumps since they are spatial in nature. More that one dump command and output file can be used during a simulation by giving

each a unique dump-ID and unique filename.

IMPORTANT NOTE: When running in parallel, unless the dump_modify sort option is invoked, the lines of per-site information written to dump files will be in an indeterminate order, i.e. not ordered by site ID. This is because the sites owned by each processor are written in a contiguous chunk. The ordering will be the same in every snapshot.

Dump snapshots will only be written on timesteps where the system time is a multiple of *delta*. Depending on now time advances in the application and solver (kinetic MC or rejection MC), the system time for a snapshot may be somewhat larger than an exact multiple of *delta*. I.e. SPPARKS will trigger the snapshot on the first timestep that the system time advances to a value >= a new delta interval.

Note that this means snapshots will not be written at the beginning or very end of a run, if the system time is not a multiple of delta. If multiple runs are performed, the same snapshot will not be written at the end of one run and the beginning of the next.

The dump_modify command can be used to alter the times at which snapshots are written out as well as define a subset of sites to write out. See the *delay*, *delta*, *logfreq*, *loglinfreq*, and *tol* keywords of the dump_modify command for details.

For the *text* format file, each snapshot begins with lines like these:

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

The first field "100" denotes which snapshot it is, numbered as 0,1,2,etc. Snapshot 0 is thus typically for the state of the system before the first run command. The second field "3.23945" is the simualtion time when the snapshot is generated.

IMPORTANT NOTE: The second simulation time field is an addition to the standard LAMMPS-style header for each snapshot.

The next lines are like these:

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

The word "ATOMS" is LAMMPS syntax, but simply means the number of sites in a SPPARKS simulation. The number "314159" will reflect any reduction in dumped site count due to the dump_modify command.

The next lines are like these:

```
ITEM: BOX BOUNDS
0 50
0 50
0 50
```

which denote the simulation box size in x,y,z. E.g, the last line is zlo and zhi.

The next line is like this:

ITEM: ATOMS id type x y z

which begins the per-site information. One line per site follows. The trailing "id type x y z" are labels for the per-site columns, using the requested fields in the dump command. The word "site" is converted to "type" so as to be compatible with how LAMMPS-style dump files are visualized. The LAMMPS default is to use the "type" value to color the object (e.g. a sphere) drawn at each site.

For the *sites* format file, each snapshot begins with lines like these.

```
Site file written by dump sites 2 command at time: 3 3.01

3 dimension
1000 sites
id site columns
0 10 xlo xhi
0 10 ylo yhi
0 10 zlo zhi
```

This is followed by a "Values" section of per-site info, with one line per site. Each line begins with a site ID, followed by the per-site values listed in the "columns" header line

See the "read_sites" command for more explanation of this format. The two time fields at the end of the first (comment) line are the same TIME info described above the the *text* style format. The "id site" keywords that preceed "columns" define what per-site values are included in the file. The keyword "id" must be the first value in each per-site line. One or more per-site values can follow. Note that it only makes sense to include the "site" or "iN" or "dN" fields as output values, since the read_sites command can only process those as input.

IMPORTANT NOTE: For this style, a filename with the "*" wildcard must be used so that a different file is written for each snapshot. The is because the read_sites command only reads a file with a single snapshot.

IMPORTANT NOTE: This style of dump command will not write "Sites" or "Neighbors" sections to the sites file. When using the sites file to continue a simulation, it is assumed that the restart script will define the sites and their neighbors in an alternate way, e.g. via the "create_box" and "create_sites" commands. Or by reading a separate sites file with that information via an earlies "read sites" command.

IMPORTANT NOTE: You must write information for all sites to the *sites* style dump file. E.g. you cannot use the dump_modify command to limit the output to a subset of sites. This is because the read_sites requires information for all sites in the system.

For the *vtk* format file, each snapshot is wrapped with a VTK-specific header and footer. Only a single field can be listed, which must be a per-site value, e.g. "site" or "iN" or "dN".

A VTK-compatible visualization program will read the information in the dump snapshot and display one object (e.g. a cube or sphere) at each point on a regular 1d or 2d or 3d lattice.

IMPORTANT NOTE: Use of the "dump_modify vtk" command is required to use this dump style. This is to make additional simulation-specific settings included in the VTK-compatible dump file.

IMPORTANT NOTE: The *vtk* style can only be used to dump sites that are on a simple, regular lattice. In 1d, this is a "line/2n" lattice. In 2d, this is a square lattice, "sq/4n" or "sq/8n". In 3d, this is a simple cubic lattice, "sc/6n" or "sc/26n". See the lattice and create_sites commands for details on these lattice types.

If a lattice command was used to create sites, then SPPARKS will check that the lattice is one of these valid styles. However, if a read_sites command was used to define sites, e.g. by reading a previous *sites*-style dump file to continue a simulation, then no lattice is defined and SPPARKS cannot check this. It is up to you to insure the

VTK output meets this restriction. Otherwise a visualization program may not be able to render a useful image.

IMPORTANT NOTE: The dump_modify sort command must be used to insure the per-site info for the regular lattice is written to the dump file in the regular ordering that VTK expects.

IMPORTANT NOTE: For this style, a filename with the "*" wildcard must be used so that a different file is written for each snapshot.

As mentioned above, the *stitch* format file is in an SQLite format.

In principle, any tool or library which reads SQLite files should be able to read a *stitch* file, but that is not recommended, For performance reasons, the Python or C API defind by the Stitch library should be used for reading and writing stitch files. SPPARKS itself reads *stitch* files using the set stitch command. Only a filename representing a single file (no wildcards) can be used with this style. A time stamp and associated SPPARKS simulation time for each snapshot is written into the SQLite file.

Note that style *stitch* can only be used for simple regular lattices. This means lattice = line (line/2n) for 1d models, square (sq/4n or sq/8n) for 2d, or simple cubic (sc/6n or sc/26n) for 3d. See the create_sites command for more details. Many of the dump_modify options are ignored for this style. Snapshots for the entire lattice are written to the file. More info about *stitch* dump files will be added to this doc page later.

Only the specified fields will be included in the dump file for each site. If no fields are listed, then a default set of fields are output, namely "id site x y z".

These are the possible field values which may be specified.

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

The *site*, *iN*, and *dN* fields specify a per-site value. *Site* is the same as *i1*. *iN* fields are integer values for integer fields 1 to N; *dN* fields are floating-point values. The application defines how many integer and floating-point values are stored for each site.

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

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

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

IMPORTANT NOTE: Not all dump styles support all the filename options described next. See the Restrictions section below for details.

Dump filenames can contain two wildcard characters. If a "*" character appears in the filename, then one file per snapshot is written and the "*" character is replaced with the timestep value. This is a counter which starts at 0, and is incremented for each snapshot. For example, tmp.dump.* becomes tmp.dump.0, tmp.dump.1, tmp.dump.2, etc. The initial value for this counter defaults to 0, but can be reset via the dump_modify first command.

If a "%" character appears in the filename, then one file is written for each processor and the "%" character is replaced with the processor ID from 0 to P-1. For example, tmp.dump.% becomes tmp.dump.0, tmp.dump.1, ...

tmp.dump.P-1, etc. This creates smaller files and can be a fast mode of output on parallel machines that support parallel I/O for output.

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

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

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

Restrictions:

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

The *stitch* style is part the STITCH package. It is only enabled if SPPARKS was built with that package. See Section 2.3 for more info on how to do this.

For the filename specified for the *sites* or *vtk* styles, a "*" wildcard must be used and a "%" wildcard cannot be used. Likewise a "*.bin" suffix cannot be used, but a "*.gz" suffix can be used.

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

Related commands:

dump_one, dump_modify, undump, stats

Default: none

dump image command

Syntax:

```
dump dump-ID image delta filename color diameter keyword value ...
```

- ID = user-assigned name for the dump
- image = style of dump command (other style *text* is discussed on the dump doc page)
- delta = time increment between dumps (seconds)
- filename = name of file to write image to
- color = attribute that determines color of each site
- diameter = attribute that determines size of each site
- zero or more keyword/value pairs may be appended
- keyword = *shape* or *sdiam* or *bdiam* or *crange* or *drange* or *size* or *view* or *center* or *up* or *zoom* or *persp* or *box* or *axes* or *shiny* or *ssao*

```
shape value = sphere or cube
  sdiam value = number = numeric value for site diameter (distance units)
  boundary values = attribute width
    attribute = attribute to use for drawing boundaries between sites
    width = width of boundary cylinders
  crange values = lo hi
     lo,hi = lower and upper bound (inclusive) of integer color attribute
  drange values = lo hi
     lo,hi = lower and upper bound (inclusive) of integer diameter attribute
  size values = width height = size of images
    width = width of image in # of pixels
    height = height of image in # of pixels
  view values = theta phi = view of simulation box
    theta = view angle from +z axis (degrees)
    phi = azimuthal view angle (degrees)
    theta or phi can be a variable (see below)
  center values = flag Cx Cy Cz = center point of image
    flag = "s" for static, "d" for dynamic
    Cx,Cy,Cz = center point of image as fraction of box dimension (0.5 = center of box)
    Cx,Cy,Cz can be variables (see below)
  up values = Ux Uy Uz = direction that is "up" in image
    Ux,Uy,Uz = components of up vector
    Ux,Uy,Uz can be variables (see below)
  zoom value = zfactor = size that simulation box appears in image
    zfactor = scale image size by factor > 1 to enlarge, factor <1 to shrink
    zfactor can be a variable (see below)
  persp value = pfactor = amount of "perspective" in image
    pfactor = amount of perspective (0 = none,  <1 = some, > 1 = highly skewed)
    pfactor can be a variable (see below)
  box values = yes/no diam = draw outline of simulation box
    yes/no = do or do not draw simulation box lines
    diam = diameter of box lines as fraction of shortest box length
  axes values = yes/no length diam = draw xyz axes
    yes/no = do or do not draw xyz axes lines next to simulation box
    length = length of axes lines as fraction of respective box lengths
    diam = diameter of axes lines as fraction of shortest box length
  shiny value = sfactor = shinyness of spheres and cylinders
    sfactor = shinyness of spheres and cylinders from 0.0 to 1.0
  ssao value = yes/no seed dfactor = SSAO depth shading
    yes/no = turn depth shading on/off
    seed = random # seed (positive integer)
    dfactor = strength of shading from 0.0 to 1.0
```

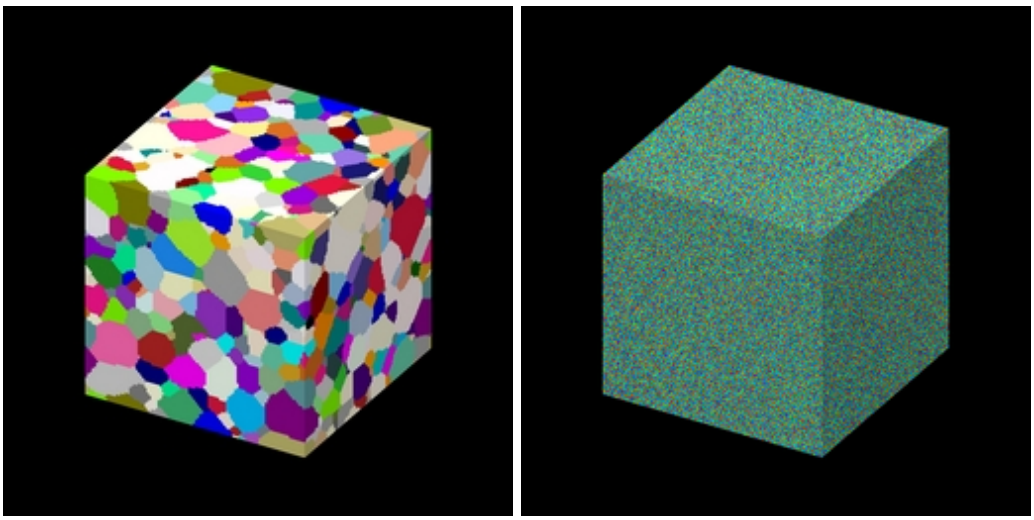
Examples:

```
dump myDump image 100 dump.*.jpg site site
dump myDump image 100 dump.*.jpg energy i2
```

Description:

Dump a high-quality ray-traced image of the sites at time intervals of *delta* during a simulation as either a JPG or PPM file. A series of such images can easily be converted into an animated movie of your simulation; see further details below. The *text* dump style writes snapshots of numerical data asociated with sites, as discussed on the dump doc page.

Here are two sample images, rendered as 1024x1024 JPG files. The left image is a million-site lattice; the right image is half a billion sites. Click to see the full-size images:



The dump_modify command can be used to alter the times at which images are written out as well as alter what sites are included in the image.

The filename suffix determines whether a JPG or PPM file is created. If the suffix is ".jpg" or ".jpeg", then a JPG file is created, else a PPM file is created, which is a text-based format. To write out JPG files, you must build SPPARKS with a JPEG library. See this section of the manual for instructions on how to do this.

Dump image filenames must contain a wildcard character "*", so that one image file per snapshot is written. The "*" character is replaced with the timestep value. For example, tmp.dump.*.jpg becomes tmp.dump.0.jpg, tmp.dump.10000.jpg, tmp.dump.20000.jpg, etc. Note that the dump_modify pad command can be used to insure all timestep numbers are the same length (e.g. 00010), which can make it easier to convert a series of images into a movie in the correct ordering.

The *color* and *diameter* settings determine the color and size of sites rendered in the image. They can be any attribute defined for the dump text command, including *site*. Note that the *diameter* setting can be overridden with a numeric value by the optional *sdiam* keyword, in which case you can specify the *diameter* setting with any valid atom attribute.

If an integer attribute such as *site* or *i2* is specified for the *color* setting, then you must use the optional *crange* keyword to specify the range of integer values that are allowed, from *lo* to *hi*. The color of each site is determined by the integer value. By default the mapping of values to colors is done by looping over the set of pre-defined colors listed with the dump_modify command, and assiging the first one to value lo, the next to value lo+1, and so on, repeating the assignment in a loop if the number of values exceeds the number of pre-defined colors. This

mapping can be changed by the dump_modify scolor command.

If a floating point attribute such as *energy* or *d1* is specified for the *color* setting, then the site's attribute will be associated with a specific color via a "color map", which can be specified via the dump_modify command. The basic idea is that the attribute will be within a range of values, and every value within the range is mapped to a specific color. Depending on how the color map is defined, that mapping can take place via interpolation so that a value of -3.2 is halfway between "red" and "blue", or discretely so that the value of -3.2 is "orange".

If an integer attribute such as *site* or *i2* is specified for the *diameter* setting, then you must use the optional *drange* keyword to specify the range of integer values that are allowed. The size of each site is determined by the integer value. By default all values has diameter 1.0. This mapping can be changed by the dump_modify sdiam command.

If a floating point attribute such as *energy* or *d1* is specified for the *diameter* setting, then the site will be rendered using the site's attribute as the diameter. If the per-site value <= 0.0, then the site will not be drawn.

The various keywords listed above control how the image is rendered. As listed below, all of the keywords have defaults, most of which you will likely not need to change. The dump modify also has options specific to the dump image style, particularly for assigning colors to atoms, bonds, and other image features.

The *shape* keyword can be specied with a value of *sphere* or *cube*, to draw either a sphere or cube at each site. Cubes typically only make sense for simple square or cubic lattices with regular spacing, so that the cubes will tile the 2d or 3d space without overlapping. The diameter specified for each site will be the diamter of the sphere or the edge length of the cube.

The *sdiam* keyword allows you to override the *diameter* setting with a specified numeric value. All sites will be drawn with that diameter.

The *boundary* keyword enables drawing of boundaries bewteen neighboring sites that have a different value of the specified attribute. This is a way to visualize the boundary between two contiguous groups of sites based on an attribute that is different for the two groups, even if the sites themselves in the 2 groups are rendered with the same color (due to the value of their *color* setting).

The specified *attribute* can be any attribute defined for the dump text command, including *site*. A boundary is only drawn between site pairs (I,J), where site I is rendered by the dump image command, site J is one of its nearest neighbors, and the value of the specified *attribute* is different for the 2 sites.

The boundary itself is drawn as 4 cylinders which outline a square. If the 2 adjacent sites are rendered as cubes (via the *shape* setting), then the square is the face common to the 2 adjacent cubes. The diameter of the cylinders is set via the *bdiam* keyword. The color of the cylinders can be set via the dump_modify boundcolor command.

The *crange* keyword must be used if the specified *color* setting is an integer attribute such as *site* or *i2*. The *lo* and *hi* values are the range of values that the attribute can have. For example, if spins in a Potts model will range from 1 to 100 (inclusive), then *lo* and *hi* should be specified as 1 and 100.

Note that internally the code allocates a vector of color values that is of length *hi*-*lo*+1. Thus you may run out of memory if *crange* encompasses N values and N is very large, e.g. 2 billion. In this case you should choose a smaller N, e.g. 10000, and use the dump_modify cwrap yes command to wrap the 2 billion possible values into N smaller values.

The *drange* keyword must be used if the specified *diameter* setting is an integer attribute such as *site* or *i2*, unless

the *sdiam* keyword is used, in which case the *diameter* setting is ignored. The *lo* and *hi* values are the range of values that the attribute can have. For example, if the *i2* attibute will take on the values -1, 0, or 1, then then *lo* and *hi* should be specified as -1 and 1

Note that internally the code allocates a vector of diameter values that is of length *hi*-*lo*+1. Thus you may run out of memory if *drange* encompasses N values and N is very large, e.g. 2 billion. In this case you should choose a smaller N, e.g. 10000, and use the dump_modify dwrap yes command to wrap the 2 billion possible values into N smaller values.

The *size* keyword sets the width and height of the created images, i.e. the number of pixels in each direction.

The *view*, *center*, *up*, *zoom*, and *persp* values determine how 3d simulation space is mapped to the 2d plane of the image. Basically they control how the simulation box appears in the image.

All of the *view*, *center*, *up*, *zoom*, and *persp* values can be specified as numeric quantities, whose meaning is explained below. Any of them can also be specified as an equal-style variable, by using v_name as the value, where "name" is the variable name. In this case the variable will be evaluated on the timestep each image is created to create a new value. If the equal-style variable is time-dependent, this is a means of changing the way the simulation box appears from image to image, effectively doing a pan or fly-by view of your simulation.

The *view* keyword determines the viewpoint from which the simulation box is viewed, looking towards the *center* point. The *theta* value is the vertical angle from the +z axis, and must be an angle from 0 to 180 degrees. The *phi* value is an azimuthal angle around the z axis and can be positive or negative. A value of 0.0 is a view along the +x axis, towards the *center* point. If *theta* or *phi* are specified via variables, then the variable values should be in degrees.

The *center* keyword determines the point in simulation space that will be at the center of the image. *Cx*, *Cy*, and *Cz* are speficied as fractions of the box dimensions, so that (0.5,0.5,0.5) is the center of the simulation box. These values do not have to be between 0.0 and 1.0, if you want the simulation box to be offset from the center of the image. Note, however, that if you choose strange values for *Cx*, *Cy*, or *Cz* you may get a blank image. Internally, *Cx*, *Cy*, and *Cz* are converted into a point in simulation space. If *flag* is set to "s" for static, then this conversion is done once, at the time the dump command is issued. If *flag* is set to "d" for dynamic then the conversion is performed every time a new image is created. If the box size or shape is changing, this will adjust the center point in simulation space.

The *up* keyword determines what direction in simulation space will be "up" in the image. Internally it is stored as a vector that is in the plane perpendicular to the view vector implied by the *theta* and *pni* values, and which is also in the plane defined by the view vector and user-specified up vector. Thus this internal vector is computed from the user-specified *up* vector as

```
up_internal = view cross (up cross view)
```

This means the only restriction on the specified *up* vector is that it cannot be parallel to the *view* vector, implied by the *theta* and *phi* values.

The *zoom* keyword scales the size of the simulation box as it appears in the image. The default *zfactor* value of 1 should display an image mostly filled by the atoms in the simulation box. A *zfactor* > 1 will make the simulation box larger; a *zfactor* < 1 will make it smaller. *Zfactor* must be a value > 0.0.

The *persp* keyword determines how much depth perspective is present in the image. Depth perspective makes lines that are parallel in simulation space appear non-parallel in the image. A *pfactor* value of 0.0 means that parallel lines will meet at infininty (1.0/pfactor), which is an orthographic rendering with no persepctive. A

pfactor value between 0.0 and 1.0 will introduce more perspective. A *pfactor* value > 1 will create a highly skewed image with a large amount of perspective.

IMPORTANT NOTE: The *persp* keyword is not yet supported as an option.

The *box* keyword determines how the simulation box boundaries are rendered as thin cylinders in the image. If *no* is set, then the box boundaries are not drawn and the *diam* setting is ignored. If *yes* is set, the 12 edges of the box are drawn, with a diameter that is a fraction of the shortest box length in x,y,z (for 3d) or x,y (for 2d). The color of the box boundaries can be set with the dump_modify boxcolor command.

The *axes* keyword determines how the coordinate axes are rendered as thin cylinders in the image. If *no* is set, then the axes are not drawn and the *length* and *diam* settings are ignored. If *yes* is set, 3 thin cylinders are drawn to represent the x,y,z axes in colors red,green,blue. The origin of these cylinders will be offset from the lower left corner of the box by 10%. The *length* setting determines how long the cylinders will be as a fraction of the respective box lengths. The *diam* setting determines their thickness as a fraction of the shortest box length in x,y,z (for 3d) or x,y (for 2d).

The *shiny* keyword determines how shiny the objects rendered in the image will appear. The *sfactor* value must be a value 0.0 <= *sfactor* <= 1.0, where *sfactor* = 1 is a highly reflective surface and *sfactor* = 0 is a rough non-shiny surface.

The *ssao* keyword turns on/off a screen space ambient occlusion (SSAO) model for depth shading. If *yes* is set, then atoms further away from the viewer are darkened via a randomized process, which is perceived as depth. The calculation of this effect can increase the cost of computing the image by roughly 2x. The strength of the effect can be scaled by the *dfactor* parameter. If *no* is set, no depth shading is performed.

A series of JPG or PPM images can be converted into a movie file and then played as a movie using commonly available tools.

Convert JPG or PPM files into an animated GIF or MPEG or other movie file:

- a) Use the ImageMagick convert program.

```
% convert *.jpg foo.gif
% convert *.ppm foo.mpg
```

- b) Use QuickTime.

  Select "Open Image Sequence" under the File menu Load the images into QuickTime to animate them
  Select "Export" under the File menu Save the movie as a QuickTime movie (*.mov) or in another format

- c) Windows-based tool.

If someone tells us how to do this via a common Windows-based tool, we'll post the instructions here.

Play the movie:

- a) Use your browser to view an animated GIF movie.

  Select "Open File" under the File menu Load the animated GIF file

- b) Use the freely available mplayer tool to view an MPEG movie.

```
% mplayer foo.mpg
```

- c) Use the Pizza.py animate tool, which works directly on a series of image files.

```
a = animate("foo*.jpg")
```

- d) QuickTime and other Windows-based media players can obviously play movie files directly.

Restrictions:

To write JPG images, you must use a -DSPPARKS_JPEG switch when building SPPARKS and link with a JPEG library. See the Making LAMMPS section of the documentation for details.

Related commands:

dump, dump_modify, undump

Default:

The defaults for the keywords are as follows:

- shape = sphere
- sdiam = not specified (use diameter setting)
- boundary = no default
- crange = no default
- drange = no default
- size = 512 512
- view = 60 30 (for 3d)
- view = 0 0 (for 2d)
- center = s 0.5 0.5 0.5
- up = 0 0 1 (for 3d)
- up = 0 1 0 (for 2d)
- zoom = 1.0
- persp = 0.0
- box = yes 0.02
- axes = no 0.0 0.0
- shiny = 1.0
- ssao = no

dump_modify command

Syntax:

```
dump_modify dump-ID keyword values ...
```

- dump-ID = ID of dump to modify
- one or more keyword/value pairs may be appended
- these keywords apply to various dump styles
- keyword = *delay* or *delta* or *fileper* or *first* or *flush* or *logfreq* or *loglinfreq* or *nfile* or *pad* or *region* or *sort* or *thresh* or *tol* or *vtk*

```
delay value = tdelay
    tdelay = delay dump until at least this time (seconds)
  delta arg = dt
    dt = time increment between dumps (seconds)
  fileper arg = Np
    Np = write one file for every this many processors
  first arg = Nfirst
    Nfirst = index of first snapshot produced, useful when restarting
  flush arg = yes or no
  logfreq or loglinfreq values = N factor
    N = number of repetitions per interval
    factor = scale factor between intervals
  nfile arg = Nf
    Nf = write this many files, one from each of Nf processors
  pad arg = Nchar = # of characters to convert timestep to
  region arg = region-ID or "none"
  sort arg = off or id or N or -N
     off = no sorting of per-site lines within a snapshot
     id = sort per-site lines by atom ID
     N = sort per-site lines in ascending order by the Nth column
     -N = sort per-site lines in descending order by the Nth column
  thresh args = attribute operation value
    attribute = same fields (id,lattice,x,etc) used by dump command
    operation = "
```

- these keywords apply only to the *image* style
- keyword = *backcolor* or *boundcolor* or *boxcolor* or *color* or *cwrap* or *dwrap* or *scolor* or *sdiam* or *smap*

```
backcolor arg = color
    color = name of color for background
  boundcolor arg = color
    color = name of color for boundaries between sites
  boxcolor arg = color
    color = name of color for box lines
  color args = name R G B
    name = name of color
    R,G,B = red/green/blue numeric values from 0.0 to 1.0
  cwrap arg = yes or no
    yes/no = do or do not wrap out-of-range color values into the defined crange
  dwrap arg = yes or no
    yes/no = do or do not wrap out-of-range diameter values into the defined drange
  scolor args = I color
    I = integer value or range of values (see below)
    color = name of color or color1/color2/... or random
  sdiam args = I diam
    I = integer value or range of values (see below)
    diam = diameter of sites of that value
```

```
smap args = lo hi style delta N entry1 entry2 ... entryN
  lo = number or min = lower bound of range of color map
  hi = number or max = upper bound of range of color map
  style = 2 letters = "c" or "d" or "s" plus "a" or "f"
    "c" for continuous
    "d" for discrete
    "s" for sequential
    "a" for absolute
    "f" for fractional
  delta = binsize (only used for style "s", otherwise ignored)
    binsize = range is divided into bins of this width
  N = # of subsequent entries
  entry = value color (for continuous style)
    value = number or min or max = single value within range
    color = name of color used for that value
  entry = lo hi color (for discrete style)
    lo/hi = number or min or max = lower/upper bound of subset of range
    color = name of color used for that subset of values
  entry = color (for sequential style)
    color = name of color used for a bin of values
```

Examples:

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

Description:

Modify the parameters of a previously defined dump command. Not all parameters are relevant to all dump styles.

These keywords apply to various dump styles, including the dump image style, except as noted below. The descriptions give details.

The *delay* keyword will suppress output until the current time is *tdelay* or greater. Note that *tdelay* is not an elapsed time since the start of the run, but an absolute time.

The *delta* keyword will reset the dump interval *delta* used in the original dump command.

The *fileper* keyword is documented below with the *nfile* keyword.

The *first* keyword can be used to set the counter used to enumerate successive snapshots. This can be useful when continuing/restarting a previous simulation, so as not to overlap new snapshots with previous output.

The counter is used in the TIMESTEP field of snapshots produced by the dump text or dump sites styles. It is also used in the filenames generated by the "*" wildcard character in the user-specified dump file name, as explained on the dump command doc page.

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

The *logfreq* and *loglinfreq* keywords will produce output at progressively larger intervals during the course of a simulation. There will be *N* outputs per interval where the size of each interval is initially *delta* and then scales up

by *factor* each time. See the stats command for more information on how the output times are specified.

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

The *nfile* or *fileper* keywords currently apply only to the *text* dump style. They can be used in conjunction with the "%" wildcard character in the specified dump file name. As explained on the dump command doc page, the "%" character causes the dump file to be written in pieces, one piece for each of P processors. By default P = the number of processors the simulation is running on. The *nfile* or *fileper* keyword can be used to set P to a smaller value, which can be more efficient when running on a large number of processors.

The *nfile* keyword sets P to the specified Nf value. For example, if Nf = 4, and the simulation is running on 100 processors, 4 files will be written, by processors 0,25,50,75. Each will collect information from itself and the next 24 processors and write it to a dump file.

For the *fileper* keyword, the specified value of Np means write one file for every Np processors. For example, if Np = 4, every 4th processor (0,4,8,12,etc) will collect information from itself and the next 3 processors and write it to a dump file.

The *pad* keyword only applies when the dump filename is specified with a wildcard "*" character which becomes the timestep. If *pad* is 0, which is the default, the timestep is converted into a string of unpadded length, e.g. 100 or 12000 or 2000000. When *pad* is specified with *Nchar* > 0, the string is padded with leading zeroes so they are all the same length = *Nchar*. For example, pad 7 would yield 0000100, 0012000, 2000000. This can be useful so that post-processing programs can easily read the files in ascending timestep order.

The *region* keyword allows sub-selection of lattice sites to output. If specified, only sites in the region will be written to the dump file or included in the image. Only one region can be applied as a filter (the last one specified). See the region command for more details. Note that a region can be defined as the "inside" or "outside" of a geometric shape, and it can be the "union" or "intersection" of a series of simpler regions.

The *sort* keyword determines whether lines of per-site output in a snapshot are sorted or not. A sort value of *off* means they will typically be written in indeterminate order, at least in parallel, since the sites are written to file in per-processor chunks. A sort value of *id* means sort the output by site ID. A sort value of N or -N means sort the output by the value in the Nth column of per-site info in either ascending or descending order.

If multiple processors are writing the dump file, via the "%" wildcard in the dump filename, then sorting cannot be performed.

IMPORTANT NOTE: Sorting dump file output requires extra overhead in terms of CPU and communication cost, as well as memory, versus unsorted output.

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

The *tol* keyword will trigger a dump snapshot if the current time is within *epsilon* of the target time for dump output.

This can be useful when running with the sweep command and the time interval per sweep leads to small

round-off differences in time. For example, if the time per sweep is 1/26 (for 26 neighbors per lattice site) and *delta* = 1.0, but an snapshot is not written at time 2.0 but at 2.0385 (0.385 = 1/26). I.e. one sweep beyond the desired dump time. Using a tol < 1/26 will give the desired snapshots at 1,2,3,4, etc.

The *vtk* keyword only applies to the *vtk* style, for which it is required. As explained on the dump vtk doc page, this style can only be used to output a single per-site value for a regular lattice of sites. The settings for this command provide information about the underlying lattice and site value bounds.

The nx,ny,nz settings are the extent of the regular lattice of sites, whether it is periodic in any dimension or not. Use a value of nz = 1 for 2d simulations and ny = nz = 1 for 1d simulations.

The minvalue and maxvalue settings are the min/max bounds within which all the per-site values for the specified single per-site field will fall. Note that the actual values do not need to extend to these bounds. E.g. the maximum initial spin value might be 1000 (e.g. for app_style potts), but at later times an individual snapshot would have no spins > 900.

These keywords apply only to the dump image style. The descriptions give details.

The *backcolor* sets the background color of the images. The color name can be any of the 140 pre-defined colors (see below) or a color name defined by the dump_modify color option.

The *boundcolor* keyword sets the color used to draw boundaries between sites, each of which is a set of 4 cylinders, as described in the dump image doc page. The color name can be any of the 140 pre-defined colors (see below) or a color name defined by the dump_modify color option.

The drawing of boundaries between neighboring sites is enabled by the *boundary* keyword of the dump image command.

The *boxcolor* keyword sets the color of the simulation box drawn around the sites in each image. See the "dump image box" command for how to specify that a box be drawn. The color name can be any of the 140 pre-defined colors (see below) or a color name defined by the dump_modify color option.

The *color* keyword allows definition of a new color name, in addition to the 140-predefined colors (see below), and associates 3 red/green/blue RGB values with that color name. The color name can then be used with any other dump_modify keyword that takes a color name as a value. The RGB values should each be floating point values between 0.0 and 1.0 inclusive.

When a color name is converted to RGB values, the user-defined color names are searched first, then the 140 pre-defined color names. This means you can also use the *color* keyword to overwrite one of the pre-defined color names with new RBG values.

The *cwrap* keyword enables wrapping of integer values used to deterimine site colors in the image, into the range specified by the *crange* keyword in the dump image command.

The *crange* keyword defines a range of values *lo* to *hi*. If the *cwrap* argument is *no*, which is the default, then values outside the range *lo* to *hi* are clipped to that range. I.e. values < *lo* become *lo* and values > *hi* become hi. This means out-of-range values will all be drawn with either the *lo* or *hi* color, which may not be what you want.

If the *cwrap* argument is *yes*, then values outside the range *lo* to *hi* are wrapped back into the range. E.g. a value of hi+1 becomes lo, hi+2 becomes lo+1, etc. Similarly, a value of lo-1 becomes hi, lo-2 becomes hi-1, etc. This is

a way to map a huge number N of possible integer values into a smaller number of M *crange* colors. This may be required if N = 2 billion spin values, since memory for that many colors cannot be allocated. Using a *crange* with M = 10000 would work in that scenario.

The *dwrap* keyword enables wrapping of integer values used to deterimine site diameters in the image, into the range specified by the *drange* keyword in the dump image command. It's purpose and the way it operates on diamenters is exactly the same as how the *cwrap* keyword operates of colors, as described above.

The *scolor* keyword can be used with the dump image command, when its site color setting is an integer attribute, and a *crange* setting from *lo* to *hi* has been specified to set the color associated with each integer value.

The specified *I* value should be an integer from *lo* to *hi* inclusive. A wildcard asterisk can be used in place of or in conjunction with the *type* argument to specify a range of values. This takes the form "*" or "*n" or "n*" or "m*n". An asterisk with no numeric values means all values from *lo* to *hi*. A leading asterisk means all values from *lo* to n (inclusive). A trailing asterisk means all values from n to *hi* (inclusive). A middle asterisk means all values from m to n (inclusive).

The specified *color* can be a single color which is any of the 140 pre-defined colors (see below) or a color name defined by the dump_modify color option. Or it can be two or more colors separated by a "/" character, e.g. red/green/blue. In the former case, that color is assigned to all the specified integer values. In the latter case, the list of colors are assigned in a round-robin fashion to each of the specified integer values.

The specified *color* can also be the word *random*. In this case, random red/blue/green color values, each from 0.0 to 1.0, are generated for each *I* value. This is a convenient way to assign a large number of random colors, without having to list them explicitly by name.

The *sdiam* keyword can be used with the dump image command, when its site diameter setting is an integer attribute, and a *drange* setting from *lo* to *hi* has been specified to set the diameter associated with each integer value. The specified *I* value should be an integer from *lo* to *hi*. As with the *scolor* keyword, a wildcard asterisk can be used as part of the *I* argument to specify a range of values.

The *smap* keyword can be used with the dump image command, when its site color setting is a floating point attribute, to setup a color map. The color map is used to assign a specific RGB (red/green/blue) color value to an individual site when it is drawn, based on the atom's attribute, which is a numeric value, e.g. its x coordinate, if the attribute "x" was specified.

The basic idea of a color map is that the site-attribute will be within a range of values, and that range is associated with a series of colors (e.g. red, blue, green). An sites's specific value (x = -3.2) can then mapped to the series of colors (e.g. halfway between red and blue), and a specific color is determined via an interpolation procedure.

There are many possible options for the color map, enabled by the *smap* keyword. Here are the details.

The *lo* and *hi* settings determine the range of values allowed for the site attribute. If numeric values are used for *lo* and/or *hi*, then values that are lower/higher than lo/hi are set to either *lo* or *hi*. I.e. the range is static. If *lo* is specified as *min* or *hi* as *max* then the range is dynamic, and the lower and/or upper bound will be calculated each time an image is drawn, based on the set of sites being visualized.

The *style* setting is two letters, such as "ca". The first letter is either "c" for continuous, "d" for discrete, or "s" for sequential. The second letter is either "a" for absolute, or "f" for fractional.

A continuous color map is one in which the color changes continuously from value to value within the range. A discrete color map is one in which discrete colors are assigned to sub-ranges of values within the range. A

sequential color map is one in which discrete colors are assigned to a sequence of sub-ranges of values covering the entire range.

An absolute color map is one in which the values to which colors are assigned are specified explicitly as values within the range. A fractional color map is one in which the values to which colors are assigned are specified as a fractional portion of the range. For example if the range is from -10.0 to 10.0, and the color red is to be assigned to atoms with a value of 5.0, then for an absolute color map the number 5.0 would be used. But for a fractional map, the number 0.75 would be used since 5.0 is 3/4 of the way from -10.0 to 10.0.

The *delta* setting is only specified if the style is sequential. It specifies the bin size to use within the range for assigning consecutive colors to. For example, if the range is from -10.0 to 10.0 and a *delta* of 1.0 is used, then 20 colors will be assigned to the range. The first will be from -10.0 <= color1 < -9.0, then 2nd from -9.0 <= color2 < -8.0, etc.

The *N* setting is how many entries follow. The format of the entries depends on whether the color map style is continuous, discrete or sequential. In all cases the *color* setting can be any of the 140 pre-defined colors (see below) or a color name defined by the dump_modify color option.

For continuous color maps, each entry has a *value* and a *color*. The *value* is either a number within the range of values or *min* or *max*. The *value* of the first entry must be *min* and the *value* of the last entry must be *max*. Any entries in between must have increasing values. Note that numeric values can be specified either as absolute numbers or as fractions (0.0 to 1.0) of the range, depending on the "a" or "f" in the style setting for the color map.

Here is how the entries are used to determine the color of an individual site, given the value X of its site attribute. X will fall between 2 of the entry values. The color of the site is linearly interpolated (in each of the RGB values) between the 2 colors associated with those entries. For example, if X = -5.0 and the 2 surrounding entries are "red" at -10.0 and "blue" at 0.0, then the site's color will be halfway between "red" and "blue", which happens to be "purple".

For discrete color maps, each entry has a *lo* and *hi* value and a *color*. The *lo* and *hi* settings are either numbers within the range of values or *lo* can be *min* or *hi* can be *max*. The *lo* and *hi* settings of the last entry must be *min* and *max*. Other entries can have any *lo* and *hi* values and the sub-ranges of different values can overlap. Note that numeric *lo* and *hi* values can be specified either as absolute numbers or as fractions (0.0 to 1.0) of the range, depending on the "a" or "f" in the style setting for the color map.

Here is how the entries are used to determine the color of an individual site, given the value X of its site attribute. The entries are scanned from first to last. The first time that *lo* <= X <= *hi*, X is assigned the color associated with that entry. You can think of the last entry as assigning a default color (since it will always be matched by X), and the earlier entries as colors that override the default. Also note that no interpolation of a color RGB is done. All sites will be drawn with one of the colors in the list of entries.

For sequential color maps, each entry has only a *color*. Here is how the entries are used to determine the color of an individual site, given the value X of its site attribute. The range is partitioned into N bins of width *binsize*. Thus X will fall in a specific bin from 1 to N, say the Mth bin. If it falls on a boundary between 2 bins, it is considered to be in the higher of the 2 bins. Each bin is assigned a color from the E entries. If E < N, then the colors are repeated. For example if 2 entries with colors red and green are specified, then the odd numbered bins will be red and the even bins green. The color of the site is the color of its bin. Note that the sequential color map is really a shorthand way of defining a discrete color map without having to specify where all the bin boundaries are.

Restrictions:

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

Related commands:

dump, dump image

Default:

The option defaults are

- backcolor = black
- boundcolor = white
- boxcolor = yellow
- color = 140 color names are pre-defined as listed below
- cwrap = no
- delay = 0.0
- delta = value for delta used in the dump command
- dwrap = no
- flush = yes
- logfreq = off
- loglinfreq = off
- pad = 0
- region = none
- scolor = * c1/c2/.../c140 where c1-c140 are the names of the 140 pre-defined colors
- sdiam = * 1.0
- smap = min max cf 2 min blue max red
- thresh = none
- tol = 0.0

These are the 140 colors that SPPARKS pre-defines for use with the dump image and dump_modify commands. Additional colors can be defined with the dump_modify color command. The 3 numbers listed for each name are the RGB (red/green/blue) values. Divide each value by 255 to get the equivalent 0.0 to 1.0 value.

aliceblue = 240, 248, 255	antiquewhite = 250, 235, 215	aqua = 0, 255, 255	aquamarine = 127, 255, 212	azure = 240, 255, 255
beige = 245, 245, 220	bisque = 255, 228, 196	black = 0, 0, 0	blanchedalmond = 255, 255, 205	blue = 0, 0, 255
blueviolet = 138, 43, 226	brown = 165, 42, 42	burlywood = 222, 184, 135	cadetblue = 95, 158, 160	chartreuse = 127, 255, 0
chocolate = 210, 105, 30	coral = 255, 127, 80	cornflowerblue = 100, 149, 237	cornsilk = 255, 248, 220	crimson = 220, 20, 60
cyan = 0, 255, 255	darkblue = 0, 0, 139	darkcyan = 0, 139, 139	darkgoldenrod = 184, 134, 11	darkgray = 169, 169, 169
darkgreen = 0, 100, 0	darkkhaki = 189, 183, 107	darkmagenta = 139, 0, 139	darkolivegreen = 85, 107, 47	darkorange = 255, 140, 0
darkorchid = 153, 50, 204	darkred = 139, 0, 0	darksalmon = 233, 150, 122	darkseagreen = 143, 188, 143	darkslateblue = 72, 61, 139
darkslategray = 47, 79, 79	darkturquoise = 0, 206, 209	darkviolet = 148, 0, 211	deeppink = 255, 20, 147	deepskyblue = 0, 191, 255

dimgray = 105, 105, 105	dodgerblue = 30, 144, 255	firebrick = 178, 34, 34	floralwhite = 255, 250, 240	forestgreen = 34, 139, 34
fuchsia = 255, 0, 255	gainsboro = 220, 220, 220	ghostwhite = 248, 248, 255	gold = 255, 215, 0	goldenrod = 218, 165, 32
gray = 128, 128, 128	green = 0, 128, 0	greenyellow = 173, 255, 47	honeydew = 240, 255, 240	hotpink = 255, 105, 180
indianred = 205, 92, 92	indigo = 75, 0, 130	ivory = 255, 240, 240	khaki = 240, 230, 140	lavender = 230, 230, 250
lavenderblush = 255, 240, 245	lawngreen = 124, 252, 0	lemonchiffon = 255, 250, 205	lightblue = 173, 216, 230	lightcoral = 240, 128, 128
lightcyan = 224, 255, 255	lightgoldenrodyellow = 250, 250, 210	lightgreen = 144, 238, 144	lightgrey = 211, 211, 211	lightpink = 255, 182, 193
lightsalmon = 255, 160, 122	lightseagreen = 32, 178, 170	lightskyblue = 135, 206, 250	lightslategray = 119, 136, 153	lightsteelblue = 176, 196, 222
lightyellow = 255, 255, 224	lime = 0, 255, 0	limegreen = 50, 205, 50	linen = 250, 240, 230	magenta = 255, 0, 255
maroon = 128, 0, 0	mediumaquamarine = 102, 205, 170	mediumblue = 0, 0, 205	mediumorchid = 186, 85, 211	mediumpurple = 147, 112, 219
mediumseagreen = 60, 179, 113	mediumslateblue = 123, 104, 238	mediumspringgreen = 0, 250, 154	mediumturquoise = 72, 209, 204	mediumvioletred = 199, 21, 133
midnightblue = 25, 25, 112	mintcream = 245, 255, 250	mistyrose = 255, 228, 225	moccasin = 255, 228, 181	navajowhite = 255, 222, 173
navy = 0, 0, 128	oldlace = 253, 245, 230	olive = 128, 128, 0	olivedrab = 107, 142, 35	orange = 255, 165, 0
orangered = 255, 69, 0	orchid = 218, 112, 214	palegoldenrod = 238, 232, 170	palegreen = 152, 251, 152	paleturquoise = 175, 238, 238
palevioletred = 219, 112, 147	papayawhip = 255, 239, 213	peachpuff = 255, 239, 213	peru = 205, 133, 63	pink = 255, 192, 203
plum = 221, 160, 221	powderblue = 176, 224, 230	purple = 128, 0, 128	red = 255, 0, 0	rosybrown = 188, 143, 143
royalblue = 65, 105, 225	saddlebrown = 139, 69, 19	salmon = 250, 128, 114	sandybrown = 244, 164, 96	seagreen = 46, 139, 87
seashell = 255, 245, 238	sienna = 160, 82, 45	silver = 192, 192, 192	skyblue = 135, 206, 235	slateblue = 106, 90, 205
slategray = 112, 128, 144	snow = 255, 250, 250	springgreen = 0, 255, 127	steelblue = 70, 130, 180	tan = 210, 180, 140
teal = 0, 128, 128	thistle = 216, 191, 216	tomato = 253, 99, 71	turquoise = 64, 224, 208	violet = 238, 130, 238
wheat = 245, 222, 179	white = 255, 255, 255	whitesmoke = 245, 245, 245	yellow = 255, 255, 0	yellowgreen = 154, 205, 50

dump_one command

Syntax:

```
dump_one dump-ID
```

- dump-ID = ID of previously defined dump

Examples:

```
dump_one mine
dump_one 2
```

Description:

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

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

Restrictions: none

Related commands:

dump

Default: none

echo command

Syntax:

```
echo style
```

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

Examples:

```
echo both
echo log
```

Description:

This command determines whether SPPARKS echoes each input script command to the screen and/or log file as it is read and processed. If an input script has errors, it can be useful to look at echoed output to see the last command processed.

Restrictions: none

Related commands: none

Default:

```
echo log
```

ecoord command

Syntax:

```
ecoord N eng
```

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

Examples:

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

Description:

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

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

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

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

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

Restrictions:

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

Related commands:

deposition, barrier

Default:

Energy values for all coordination numbers are set to 0.

ellipsoid_depth command

Syntax:

```
ellipsoid_depth D
```

- D = Maximum depth

Examples:

```
ellipsoid_depth 30
```

Description:

This command is used in the ellipsoid mode of the potts/weld_jom application to define the maximum ellipsoid depth.

It is also used in the keyhole mode of the potts/weld_jom application to define the maximum depth of the shallow ellipsoid.

Restrictions:

This command can only be used as part of the app_style potts/weld_jom application.

It must be a positive value.

Related commands:

deep_length, deep_width

Default: 1/4 * zhi

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 neighors 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 atoms must be on those sites in order for the event to potentially 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 via the formula:

```
rate = exp(-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 atoms will be on which sites 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 a reaction/diffusion model for hydrogen interstitials in an erbium lattice.

Restrictions: none

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

Related commands:

app_style erbium

Default: none

event_ratios command

Syntax:

```
event_ratios Rgg Rpm Rv
```

- Rgg = value of number of attempts for grain growth event in Monte Carlo simulation of sintering
- Rpm = value of number of attempt for pore migration event in Monte Carlo simulation of sintering
- Rv = value of number of attempts for vacancy creation and annihilation event in Monte Carlo simulation of sintering

Examples:

```
event_ratios 2.0 1.0 4.0
```

Description:

This command sets the number of attempts for each event in the sintering application. Each event is attempted with a frequency proportional to the ratio between the particular number of attempts given and the sum of the number of attempts for all the events. The typical usage would be to alter the frequency of occurrence of the events. The events correspond to: grain growth, pore migration and vacancy creation and annihilation.

Restrictions: these should be positive values.

This command can only be used as part of the sintering application. See the doc pages for the sintering application defined by the app_style sinter command for further details.

Related commands:

event_temperatures

Default:

The default event ratios are 1.0 1.0 1.0.

event_temperatures command

Syntax:

```
event_temperatures Tgg Tpm Tv
```

- Tgg = value of temperature for grain growth in Monte Carlo simulation of sintering
- Tpm = value of temperature for pore migration in Monte Carlo simulation of sintering
- Tv = value of temperature for vacancy creation and annihilation in Monte Carlo simulation of sintering

Examples:

```
event_temperatures 2.0 1.0 15.0
```

Description:

This command sets the event temperature as used in the sintering application. The typical would be as part of a Boltzmann factor that alters the probabilities of event acceptance and rejection.

Restrictions: these should be positive values.

This command can only be used as part of the sintering application. See the doc pages for the sintering application defined by the app_style sinter command for further details.

Related commands:

event_ratios

Default:

The default temperatures are 1.0 1.0 15.0.

if command

Syntax:

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

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

Examples:

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

Description:

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

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

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

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

Restrictions: none

Related commands:

variable

Default: none

include command

Syntax:

```
include file
```

- file = filename of new input script to switch to

Examples:

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

Description:

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

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

Restrictions: none

Related commands:

variable, jump

Default: none

inclusion command

Syntax:

```
inclusion x y z r
```

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

Examples:

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

Description:

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

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

Restrictions:

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

Related commands:

app_style membrane

Default: none

jump command

Syntax:

```
jump file label
```

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

Examples:

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

Description:

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

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

```
variable a loop 10
label loop
run 5.0
next a
jump in.lj loop
```

If the jump *file* argument is a variable, the jump command can be used to cause different processor partitions to run different input scripts. In this example, SPPARKS is run on 40 processors, with 4 partitions of 10 procs each. An in.file containing the example variable and jump command will cause each partition to run a different simulation.

```
mpirun -np 40 lmp_ibm -partition 4x10 -in in.file

variable f world script.1 script.2 script.3 script.4
jump $f
```

Restrictions:

If you jump to a file and it does not contain the specified label, SPPARKS will come to the end of the file and exit.

Related commands:

variable, include, label, next

Default: none

label command

Syntax:

```
label ID
```

- ID = string used as label name

Examples:

```
label xyz
label loop
```

Description:

Label this line of the input script with the chosen ID. Unless a jump command was used previously, this does nothing. But if a jump command was used with a label argument to begin invoking this script file, then all command lines in the script prior to this line will be ignored. I.e. execution of the script will begin at this line. This is useful for looping over a section of the input script as discussed in the jump command.

Restrictions: none

Related commands: none

Default: none

lattice command

Syntax:

```
lattice style args
```

- style = *none* or *line/2n* or *sq/4n* or *sq/8n* or *tri* or *sc/6n* or *sc/26n* or *bcc* or *fcc* or *diamond* or *fcc/octa/tetra* or *random/1d* or *random/2d or random/3d*

```
none args: none
  all other styles except random = scale
    scale = lattice constant (distance units)
  random/1d args = Nrandom cutoff
  random/2d args = Nrandom cutoff
  random/3d args = Nrandom cutoff
    Nrandom = # of random sites
    cutoff = distance within which sites are connected (distance units)
```

Examples:

```
lattice sq/4n 1.0
lattice fcc 3.52
lattice random/3d 10000 2.0
lattice none
```

Description:

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

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

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

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

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

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

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

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

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

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

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

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

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

- neighbors of each fcc site: 12 fcc, 6 octa, 8 tetra
- neighbors of each octa site: 6 fcc, 12 octa, 8 tetra
- neighbors of each tetra site: 4 fcc, 4 octa, 6 tetra

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

The command "lattice none" can be used to turn off a previous lattice definition. Any command that attempts to use the lattice directly will then generate an error. No additional arguments need be used with "lattice none".

Restrictions: none

Related commands:

dimension, create_sites, region

Default: none

log command

Syntax:

```
log file
```

- file = name of new logfile

Examples:

```
log log.equil
```

Description:

This command closes the current SPPARKS log file, opens a new file with the specified name, and begins logging information to it. If the specified file name is *none*, then no new log file is opened.

If multiple processor partitions are being used, the file name should be a variable, so that different processors do not attempt to write to the same log file.

The file "log.spparks" is the default log file for a SPPARKS run. The name of the initial log file can also be set by the command-line switch -log. See this section for details.

Restrictions: none

Related commands: none

Default:

The default SPPARKS log file is named log.spk

next command

Syntax:

```
next variables
```

- variables = one or more variable names

Examples:

```
next x
next a t x myTemp
```

Description:

This command is used with variables defined by the variable command. It assigns the next value to the variable from the list of values defined for that variable by the variable command. Thus when that variable is subsequently substituted for in an input script command, the new value is used.

See the variable command for info on how to define and use different kinds of variables in SPPARKS input scripts. If a variable name is a single lower-case character from "a" to "z", it can be used in an input script command as $a or $z. If it is multiple letters, it can be used as ${myTemp}.

If multiple variables are used as arguments to the *next* command, then all must be of the same variable style: *index*, *loop*, *universe*, or *uloop*. An exception is that *universe*- and *uloop*-style variables can be mixed in the same *next* command. *Equal*- or *world*-style variables cannot be incremented by a next command. All the variables specified are incremented by one value from their respective lists.

When any of the variables in the next command has no more values, a flag is set that causes the input script to skip the next jump command encountered. This enables a loop containing a next command to exit.

When the next command is used with *index*- or *loop*-style variables, the next value is assigned to the variable for all processors. When the next command is used with *universe*- or *uloop*-style variables, the next value is assigned to whichever processor partition executes the command first. All processors in the partition are assigned the same value. Running SPPARKS on multiple partitions of processors via the "-partition" command-line switch is described in this section of the manual. *Universe*- and *uloop*-style variables are incremented using the files "tmp.spparks.variable" and "tmp.spparks.variable.lock" which you will see in your directory during such a SPPARKS run.

Here is an example of running a series of simulations using the next command with an *index*-style variable. If this input script is named in.polymer, 8 simulations would be run using data files from directories run1 thru run8.

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

If the variable "d" were of style *universe*, and the same in.polymer input script were run on 3 partitions of processors, then the first 3 simulations would begin, one on each set of processors. Whichever partition finished first, it would assign variable "d" the 4th value and run another simulation, and so forth until all 8 simulations were finished.

Jump and next commands can also be nested to enable multi-level loops. For example, this script will run 15 simulations in a double loop.

```
variable i loop 3
variable j loop 5
clear
...
read_data data.polymer.$i$j
print Running simulation $i.$j
run 10000
next j
jump in.script
next i
jump in.script
```

Restrictions: none

Related commands:

jump, include, shell, variable,

Default: none

pair_coeff command

Syntax:

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

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

Examples:

Examples:

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

Description:

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

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

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

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

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

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

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

- pair_style lj/cut - cutoff Lennard-Jones potential

Restrictions: none

Related commands:

pair_style

Default: none

pair_style lj command

Syntax:

```
pair_style lj Ntypes cutoff
```

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

Examples:

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

Description:

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

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

Rc is the cutoff.

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

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

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

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

Mixing info:

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

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

Restrictions: none

Related commands: none

Default: none

pair_style command

Syntax:

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

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

Examples:

```
pair_style lj 1 2.5
```

Description:

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

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

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

- pair_style lj/cut - cutoff Lennard-Jones potential

Restrictions: none

Related commands:

pair_style

Default: none

pin command

Syntax:

```
pin fraction multiflag nthresh
```

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

Examples:

```
pin 0.1 0 2
```

Description:

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

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

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

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

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

Restrictions: none

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

Related commands:

app_style potts/pin

Default: none

potts/am/bezier command

Syntax:

```
potts/am/bezier keyword args
```

- keyword = *control_points x* or *y* or *z* or *beta*

```
control_points x args = P0x P1x P2x P3x P4x
    P0x P1x P2x P3x P4x x component values for 5 control points (floating)
  control_points y args =      P1y P2y P3y
    P1y P2y P3y = y component values for 3 control points (floating), app automatically
  control_points z args =       P1z P2z P3z
    P1z P2z P3z = z component values for 3 control points (floating), app automatically
  beta args = betay, betaz
    betay, betaz = lateral cross-section weights effecting lateral convexity
```

Examples:

```
potts/am/bezier control_points x -6.9 -6.9   0.6  6.9 6.9
potts/am/bezier control_points y       0.8   2.1  3.8
potts/am/bezier control_points z      -0.9, -1.0 -2.8
potts/am/bezier beta 1.0 0.5
```

Description:

This command is defined and used by the app_style potts/am/bezier application.

The command argument *control_points* is required 3 times for specification of *x,y,z* component values of control points used to define bezier melt pool surface.

The command argument *beta* is optional and can be used to adjust surface convexity according to schematic image shown above. Note that app calculates distance to melt pool surface using a closest point projection algorithm; this calculation is robust for convex surfaces but can and probably will fail for concave surfaces -- results may vary.

In the example above, 5 required floating point values are input for *x* component of control points, and 3 floating point values are input for *y,z* components. Values for *beta* are the default values.

Restrictions:

This command can only be used with the app_style potts_am_bezier application.

Related commands:

app_style potts_am_bezier

Default:

There are no defaults for *control_point* values and they must be supplied as part of script running the potts_am_bezier app. The *beta* keyword command is optional however and the above example shows the default values *betay=1.0,betaz=0.5*.

print command

Syntax:

```
print string
```

- string = text string to print. may contain variables

Examples:

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

Description:

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

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

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

Restrictions: none

Related commands:

variable

Default: none

processors command

Syntax:

```
processors Px Py Pz
```

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

Examples:

```
processors 2 4 4
```

Description:

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

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

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

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

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

Restrictions:

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

Related commands: none

Default:

SPPARKS chooses Px, Py, Pz

pulse command

Syntax:

```
pulse A period
```

- A = fractional amplification of potts/weld model pool size
- period = cyclic time period expressed in Monte Carlo steps (MCS)

Examples:

```
pulse 0.25 64
```

Description:

This command defines an optional pulsed power simulation to the app_style potts/weld application. The parameter *A* scales up the weld pool size reaching a maximum size proportional to *(1+A)*. *A* must be > 0.0. To simulate the pulsed aspect of the model, a time *period* is specified. *Period* must be > 2.0.

Note that this command generally produces a spatially periodic effect that also depends upon the *velocity* parameter in the app_style potts/weld application.

Restrictions:

This command can only be used as part of the app_style potts/weld application.

Related commands:

app_style potts/weld

Default: none

If this command is not present in a weld simulation, then the pulse aspect of the app_style potts/weld application is not active.

read_sites command

Syntax:

```
read_sites file
```

- file = name of data file to read in

Examples:

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

Description:

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

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

The read_sites command can be used in one of 3 scenarios:

If a simulation box has not already been created and no event sites exist, then the data file defines the box size (in the header), and it must define Sites. It must also define Neighbors for on-lattice applications. The Values section is optional, since these can be set later via the set command.

If a simulation box has already been defined (by the "create_box" command), but no sites have previously been defined, then the data file must define Sites. It must also define Neighbors for on-lattice applications. The Values section is optional. If the data file defines a box size, it must be consistent with the simulation box that already exists.

If a simulation box has already been defined, and sites have previosly been defined (by the "create_sites" command or a previous read_sites command), then no Sites or Neighbors can be specfied, but the Values section is used to set all or a subset of the per-site values defined by the application. This is a means of continuing a previous simulation using a file written by the dump sites command as a restart file, since it writes in the format that this command reads.

Note that the periodicity of the simulation box, as defined by the boundary command is not considered by this command when defining sites or neighbors. It is up to you to insure sites are not duplicated on a periodic boundary, or that a site's neighbor list does not include sites that are on the other side of the simulation box when the boundary is not periodic. This is in contrast to the create_sites command which accounts for both of these issues when defining sites and their neighbors.

The first line of the header of the data file is always skipped; it typically contains a description of the file. Then lines are read one at a time. Lines can have a trailing comment starting with '#' that is ignored. If the line is blank (only whitespace after comment is deleted), it is skipped. If the line contains a header keyword, the corresponding

value(s) is read from the line. If it doesn't contain a header keyword, the line begins the body of the file.

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

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

These are the recognized header keywords. Header lines can come in any order. The value(s) are read from the beginning of the line. Thus the keyword *sites* should be in a line like "1000 sites"; the keyword *ylo yhi* should be in a line like "-10.0 10.0 ylo yhi". All these numeric settings have a default value of 0, except the lo/hi box size defaults which are -0.5 and 0.5. A line need only appear if the value is different than the default. If the keyword values have already been defined (e.g. box sizes for a previously created simulation box), then the values in the data file must match.

- *dimension* = dimension of system = 1,2,3
- *sites* = number of sites
- *max neighbors* = max # of neighbors of any site
- *label1 label2 ... labelN values* = column labels for Values section
- *xlo xhi* = simulation box boundaries in x dimension
- *ylo yhi* = simulation box boundaries in y dimension
- *zlo zhi* = simulation box boundaries in z dimension

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

The *values* setting is only needed if a Values section is included in the file, and if it does not list per-site info for all the integer and floating point values defined by the application. If only a subset of per-site values are listed in each line, then the *values* setting labels what they are. The labels have the same syntax as those defined by the dump sites command, namely "id", "site", "iN", or "dN". Note that "id" must always be included and come first, so that SPPARKS can assign the values that follow to the correct site.

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

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

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

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

10 10.0 5.0 6.0 # impuity site

Sites section:

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

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

- example:

```
101 7.0 0.0 3.0
```

There must be N lines in this section where N = number of sites and is defined by the *sites* keyword in the header section of the file. The lines can appear in any order.

Neighbors section:

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

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

- example:

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

There must be N lines in this section where N = number of sites and is defined by the *sites* keyword in the header section of the file. The lines can appear in any order.

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

Values section:

- one line per site
- line syntax: ID value1 value2 ...

```
ID = global site ID (1-N)
value1,value2,... = integer or floating point values for the site
```

- example:

```
101 1 3 4.0
```

There must be N lines in this section where N = number of sites and is defined by the *sites* keyword in the header section of the file. The lines can appear in any order.

The number of values per site depends on the *comment* keyword in the header section of the file. If it is not defined, then the default line syntax is assumed to be:

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

  meaning that all per-site values must be listed on each line. In the default case, they are listed in order, with the integer values first, followed by the floating-point values.

Restrictions:

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

Related commands:

create_box, create_sites, set

Default: none

region command

Syntax:

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

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

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

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

```
side value = in or out
    in = the region is inside the specified geometry
    out = the region is outside the specified geometry
```

Examples:

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

Description:

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

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

Commands which use regions typically test whether a lattice site is contained in the region or not. For this purpose, coordinates exactly on the region boundary are considered to be interior to the region. This means, for example, for a spherical region, a lattice site on the sphere surface would be part of the region if the sphere were defined with the *side in* keyword, but would not be part of the region if it were defined using the *side out* keyword. See more details on the *side* keyword below.

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

IMPORTANT NOTE: Regions in SPPARKS are always 3d geometric objects, regardless of whether the dimension of the lattice is 1d or 2d or 3d. Thus when using regions in a 2d simulation, for exapmle, you should be careful to define the region so that its intersection with the 2d x-y plane of the simulation has the 2d geometric extent you want. Also note that for 2d simulations, SPPARKS expects lattice sites to lie in the z=0 plane, and similarly for 1d (y = z = 0), so the regions you define as input to the create_box command should reflect that.

For style *cylinder*, the c1,c2 params are coordinates in the 2 other dimensions besides the cylinder axis dimension. For dim = x, c1/c2 = y/z; for dim = y, c1/c2 = x/z; for dim = z, c1/c2 = x/y. Thus the third example above specifies a cylinder with its axis in the y-direction located at x = 2.0 and z = 3.0, with a radius of 5.0, and extending in the y-direction from -5.0 to the upper box boundary.

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

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

Restrictions: none

Related commands:

lattice, create_sites

Default:

The option defaults are side = in.

reset_time command

Syntax:

reset_time style options:pre

style = *stitch* or *time*

- for style *stitch*, options = "stitch_file_name" "last" or "first"
- for style *time*, options = new time

Examples:

```
reset_time stitch outputfile.st last
reset_time stitch outputfile.st first
reset_time 0.0
reset_time 100.0
```

Description:

Set the current time to the specified value. This can be useful if a preliminary run was performed and you wish to reset the time before performing a subsequent run. For the *stitch* style option, this is particularly useful for setting the current simulation time to either the 'first' time step or 'last' time step contained in specified stitch file: 'stitch_file_name'.

Restrictions: none

Related commands: none

Default: none

run command

Syntax:

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

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

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

Examples:

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

Description:

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

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

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

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

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

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

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

Restrictions: none

Related commands: none

Default:

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

sector command

Syntax:

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

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

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

Examples:

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

Description:

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

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

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

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

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

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

If the *tstop* keyword is set to a value > 0.0, it sets the time per sector to the specified value. For a KMC algorithm, events are performed until this time threshhold is reached. The final event, whose time >= tstop, is not accepted. For a rKMC algorithm, the time per attempted event = dt_sweep is defined by the application, and the number of attempted events in each sector is set to nsite*int(tstop/dt_sweep). Because of integer truncation, the simulation time increment in rKMC may differ slightly from the specified tstop.

If the *nstop* keyword is set to a value > 0.0, it sets the average number of events (or attempts) per site. For example, an *nstop* value of 2.0 means attempt 2 events per site for a rKMC algorithm. For a KMC algorithm, this is converted into a time using pmax = the maximum propensity per site. At the start of each visit to a sector, the per-site propensity for the sector = psect, is computed. Psect is the total propensity of the sector divided by the total number of active sites, which are those with propensity greater than zero. After all sectors have been visited, pmax is set to the largest value of psect across all processors and sectors, and the threshold time for the next visit to each sector is set to nstop/pmax.

In the KMC case, this means that if the total propensity of the system decreases as the simulation proceeds (e.g. grain growth occurs), then the effective time per sweep will increase in an adaptive way. Said another way, the number of events per sweep will remain roughly constant, as the time per event increases. In the rKMC case, the time per attempt is constant due to the use of a null-bin, so there is no adaptivity.

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

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

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

Restrictions:

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

Related commands:

app_style, solve_style, sweep

Default:

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

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

seed command

Syntax:

```
seed Nvalue
```

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

Examples:

```
seed 5838959
```

Description:

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

Restrictions: none

Related commands: none

Default: none

set command

Syntax:

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

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

```
value arg = nvalue
    nvalue = value to set sites to
  range args = lo hi
    lo,hi = range of values to set sites to
  unique args = none
  displace arg = delta
     delta = max distance to displace the site
  stitch args = stitchfile tstamp
     stitchfile = name of STITCH file
     tstamp = first or last or a floating point value
     if first: then site values from first timestamp in stitch file are read in
     if last: then site values from last timestamp in stitch file are read in
     if floating point value: site values for this timestamp are read in

bfile args = bfilename
     bfilename = name of binary file
```

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

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

Examples:

```
set i1 value 2 fraction 0.5
set d1 range 1.0 2.0 loop local
set xyz displace 0.2
set i1 range 1 50 if x <20 if i2 = 3
set site stitch equiaxed.st first
set site stitch equiaxed.st last
set site stitch equiaxed.st 1.0
set i1 stitch equiaxed.st 1.0
set d1 stitch equiaxed.st 1.0
```

Description:

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

be changed, either before the first run, or between runs.

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

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

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

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

For style *unique*, the per-site quantity is set to the site ID, which is effectively a value unique to each site. This can be useful, for example, for setting the initial spin of each site to a unique value.

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

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

Styles *stitch* and *bfile* can only be used for simple regular lattices. This means lattice = line (line/2n) for 1d models, square (sq/4n or sq/8n) for 2d, or simple cubic (sc/6n or sc/26n) for 3d. See the create_sites command for more details. The *fraction*, *loop*, *region* and *if* keywords are ignored for these styles; these styles set values for all sites in the system.

For style *stitch*, a *stitch* file is read to extract values associated with a specified *label*. The *stitch* file can be created by the dump stitch command or an external program. A *stitch* file can store multiple values for the same site, each with a different timestamp. A different number of values can also be associated with each site. The specified *tstamp* value is used to determine which of the multiple values is used for initializing each site. The specified time value *tstamp* should exist in the file for at least some sites; it can exist on all the sites or just some of them. For sites that do not have a value for the *tstamp* time but have a value at an earlier time, those sites will be set with the value matching the most recent time stamp stored in the file. It is an error if the file does not contain any values for the specified *tstamp*. A subsequent set command can be used to initialize the value of any sites in a different way.

See the examples/stitch dir for examples of SPPARKS scripts that read and write *stitch* files.

For style *bfile*, a binary file is read to extract the values associated with the specified *label*. The binary file must be created by an external program. It should contain 3 integer header values: Nx, Ny, Nz. These must match the size of the regular lattice defined for the enitre simulation box. For 2d simulations, Nz = 1.

The file must then contain N integer or double values, depending on whether the *label* is for integer or floating point site values. N must be Nx * Ny * Nz. The site values in the file must be ordered with x varying fastest, then y, and z slowest. The binary file is read by a single processor, and the values are broadcast to all processors. Each processor then extracts the subset of values from the 3d array of sites that correspond to the sites in its sub-domain of the simulation box.

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

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

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

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

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

Restrictions:

The *stitch* style is part the STITCH package. It is only enabled if SPPARKS was built with that package. See Section 2.3 for more info on how to do this.

Related commands:

create_sites, read_sites

Default:

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

shell command

Syntax:

```
shell style args
```

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

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

Examples:

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

Description:

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

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

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

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

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

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

Restrictions:

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

Related commands: none

Default: none

app_style command

Syntax:

```
app_style style args
```

- style = application style name
- args = args

Examples:

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

Description:

This command ...

Restrictions: none

Related commands:

variable, ...

Default: none

app_style command

Syntax:

```
app_style style args
```

- style = application style name
- args = args

Examples:

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

Description:

This command ...

Restrictions: none

Related commands:

variable, ...

Default: none

solve_style command

Syntax:

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

- style = *linear* or *tree* or *group* or *none*

```
linear arg = none
  tree arg = none
  group args = hi lo
    hi,lo = range of allowed probabilities
  none arg = none
```

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

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

Examples:

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

Description:

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

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

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

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

The *group* style chooses an event using the composition and rejection (CR) algorithm described originally in Devroye and discussed in Slepoy. Its cost to pick an event scales as O(1) as it is a constant time algorithm. It requires that you bound the *hi* and *lo* probabilities for any event that will be considered with the solver. Note that for on-lattice applications this is typically the total probability of all events associated with a site. The value of *lo* must be > 0.0 and *lo* cannot be >= *hi*. For efficiency purposes it is good to choose bounds that are reasonably tight.

By default, the *group* style will create groups whose boundaries cascade downward in powers of 2 from *hi* to *lo*. I.e. the first group is from hi/2 to hi, the second group is from hi/4 to hi/2, and continuing until *lo* is reached. 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.

IMPORTANT NOTE: For the *group* style, if an event is generated that has a probability = 0.0 (e.g. a site has no possible event), that is not a violation of the *lo* bound. However if an event is generated with a non-zero probability and the probability is less than *lo* or greater than *hi*, then the probability is reset by the solver to the *lo* or *hi* bound. If this occurs during a run, SPPARKS will print out a warning message (either before the run, or at the end of the script), since it indicates events have been selected using (slightly) different probabilities than the model generated. This allows you to set a different *lo* or *hi* bound and re-run the simulation.

The *none* style deletes any KMC solver previously defined. This may be useful for transitioning from a KMC solver in one run to a sweeping method with a rejection-KMC solver in a subsequent run.

Restrictions:

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

Related commands:

app_style, sweep_style

Default: none

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

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

(Gibson) Gibson and Bruck, J Phys Chem, 104, 1876 (2000).

(Devroye) Devroye, Non-Uniform Random Variate Generation, Springer-Verlag, New York (1986).

(Slepoy) Slepoy, Thompson, Plimpton, J Chem Phys, 128, 205101 (2008).

app_style command

Syntax:

```
app_style style args
```

- style = application style name
- args = args

Examples:

```
app_style ising 100 100
app_style potts 1000 1000 4
```

Description:

This command ...

Restrictions: none

Related commands:

variable, ...

Default: none

stats command

Syntax:

```
stats delta keyword values ...
```

- delta = time increment between statistical output (seconds)
- zero or more keyword/value pairs may be appended
- keyword = *delay* or *logfreq* or *loglinfreq* or *tol*

```
  delay value = tdelay
      tdelay = delay stats until at least this time (seconds)
    logfreq or loglinfreq values = N factor
      N = number of repetitions per interval
      factor = scale factor between intervals
    tol value = epsilon
      epsilon = output stats if time is within epsilon of target time (seconds)
```

Examples:

```
stats 0.1
stats 0.1 delay 0.5
stats 1.0 loglinfreq 7 10.0
```

Description:

Print statistics to the screen and log file every so many seconds during a simulation. A value of 0.0 for delta means only print stats at the beginning and end of the run, in which case no optional keywords can be used.

The quantities printed are elapsed CPU time followed by those provided by the application, followed by those provided by any diagnostics you have defined.

Typically the application reports only the number of events or sweeps executed, followed by the simulation time, but other application-specific quantities may also be reported. Quantities such as the total energy of the system can be included in the output by creating diagnostics via the diag_style command.

The *delay* keyword will suppress output until the current time is *tdelay* or greater. Note that *tdelay* is not an elapsed time since the start of the run, but an absolute time.

Using the *logfreq* or *loglinfreq* keyword will produce statistical output at progressively larger intervals during the course of a simulation. There will be *N* outputs per interval where the size of the interval is initially *delta* and then scales up by *factor* each time. With *loglinfreq*, output times increase arithmetically within an interval; with *logfreq* the output times increase geometrically.

For example, this command

```
stats 0.1 loglinfreq 7 10.0
```

will produce output at 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 times:

```
t = 0, 0.1, 0.2, 0.4, 0.8, 1.6, ...
```

This command

```
stats 1.0 logfreq 10 10.0
```

will produce output at times:

```
t = 0, 1.0, 1.26, 1.58, 2.00, 2.51, 3.16, 3.98, 5.01, 6.31, 7.94, 10.0, ...
```

Note that in the above examples the times are the earliest times that output will be produced. In practice, because time is incremented in discrete jumps, output will be produced at times somewhat later than these times.

If *N* is specified as 0, then this will turn off logarithmic output, and revert to regular output every *delta* seconds.

The *tol* keyword will trigger output if the current time is within *epsilon* of the target time for output.

This can be useful when running with the sweep command and the time interval per sweep leads to small round-off differences in time. For example, if the time per sweep is 1/26 (for 26 neighbors per lattice site) and *delta* = 1.0, but output does not appear at time 2.0 but at 2.0385 (0.385 = 1/26). I.e. one sweep beyond the desired output time. Using a tol < 1/26 will give the desired outputs at 1,2,3,4, etc.

Restrictions:

See the doc pages for quantities provided by particular app_style and diag_style commands for further details.

Related commands:

dump, diag_style

Default:

The default delta setting is 0.0 (if this command is not used), so that stats will only be output at the beginning and end of the run. The keyword defaults are delay = 0.0, no logarithmic output, tol = 0.0.

sweep command

Syntax:

```
sweep style keyword value ...
```

- style = *random* or *raster* or *color* or *color/strict* or *none*
- zero or more keyword/value pairs may be appended
- keyword = *mask*

```
mask value = yes or no
    yes/no = mask out sites than cannot change
```

Examples:

```
sweep random
sweep raster mask yes ...
```

Description:

Use a rejection kinetic Monte Carlo (rKMC) algorithm for an on-lattice application. If rKMC is not used then a kinetic Monte Carlo (KMC) algorithm must be used as defined by the solve_style command.

The rKMC algorithm in SPPARKS selects sites on a lattice in an order determined by this command and requests that the application perform events. The application defines the geometry and connectivity of the lattice, what the possible events are, and defines their rates and acceptance/rejection criteria.

The ordering of selected sites is also affected by the sector command, which partitions each processor's portion of the simulation domain into sectors which are quadrants (2d) or octants (3d). In this case, the ordering described below is within each sector. Sectors are looped over one at a time, interleaved by communication of lattice values inbetween.

For the *random* style, sites are chosen randomly, one at a time.

For the *raster* style, a sweep of the lattice is done, as a loop over all sites in a pre-determined order, e.g. a triple loop over i,j,k for a 3d cubic lattice.

For the *color* style, lattice sites are partitioned into sub-groups or colors which are non-interacting in the sense that events on two sites of the same color can be perfored simultaneously without conflict. This enables parallelism since events on all sites of the same color can be attempted simultaneously. Similar to sectors, the colors are looped over, interleaved by communication of lattice values inbetween.

The *color/strict* style is the same as the *color* style except that random numbers are generated in a way that is independent of the processor which generates them. Thus SPPARKS should produce the same answer, independent of how many processors are used. This can be useful in debugging an application.

If the application supports it, the *mask* keyword can be set to *yes* to skip sites which cannot perform an event due to the current value of the site and its neighbors. Enabling masking should not change the answer given by a simulation (in a statistical sense); it only offers a computational speed-up. For example, sites in the interior of grains in a Potts grain-growth model may have no potential of flipping their value. Masking can only be set to *yes* if the temperature is set to 0.0, since otherwise there is a finite probability of any site performing an event.

The *none* style deletes any rKMC sweeping algorithm previously defined. This may be useful for transitioning from a rKMC solver in one run to a KMC solver in a subsequent run.

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.

time_sinter_start command

Syntax:

```
time_sinter_start tss
```

- tss = value of time to start the vacancy creation and annihilation event in Monte Carlo sintering simulation

Examples:

```
time_sinter_start 250
```

Description:

This command sets the time to start the calculation of the vacancy creation and annihilation event in the sintering application. Prior to that time the sintering simulation just entails grain growth and pore migration with virtually no densification. The typical usage would be as part of a random initialization where the grain structure should acquire certain size before attempting any densification stage.

Restrictions: this should be a positive value

This command can only be used as part of the sintering application. See the doc pages for the sintering application defined by the app_style sinter command for further details.

Related commands: none

Default:

The default time to start sintering is 50.

undump command

Syntax:

```
undump dump-ID
```

- dump-ID = ID of previously defined dump

Examples:

```
undump mine
undump 2
```

Description:

Turn off a previously defined dump command so that it is no longer active. This closes the file associated with the dump.

Restrictions: none

Related commands:

dump

Default: none

variable command

Syntax:

```
variable name style args ...
```

- name = name of variable to define
- style = *index* or *loop* or *world* or *universe* or *uloop* or *equal* or *atom*

```
index args = one or more strings
  loop args = N = integer size of loop
  world args = one string for each partition of processors
  universe args = one or more strings
  uloop args = N = integer size of loop
  equal args = one formula containing numbers, math operations, variable references
    numbers = 0.0, 100, -5.4, 2.8e-4, etc
    constants = PI
    keywords = time, nglobal
    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
Constants	PI
Keywords	time, nglobal
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

The keywords currently allowed in a formula are *time* and *nglobal*. *Time* is the current simulation time. *Nglobal* is the number of sites in the model.

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

weld_shape_ellipse command

Syntax:

```
weld_shape_ellipse width length
```

- width = principal dimension of ellipse and maximum width of the melt pool along x-axis
- length = principal dimension of ellipse and maximum length of the melt pool along y-axis

Examples:

weld_shape_ellipse 100 150

Above command specifies an elliptical weld pool shape with width and length of 100 and 150 respectively.

Description:

Specify size of elliptical shaped weld pool at top surface of weld. Shape of pool at root surface (bottom) is controlled by *alpha* in potts/weld.

Restrictions:

This command is only valid when used with potts/weld.

Related commands:

weld_shape_teardrop

Default: none

weld_shape_teardrop command

Syntax:

```
weld_shape_teardrop width w case i
```

- width w = keyword/value pair; w is pool width measured along y-axis
- case i = keyword/value pair; i takes on allowable values of I or II or III designating one of the following 3 teardrop shapes

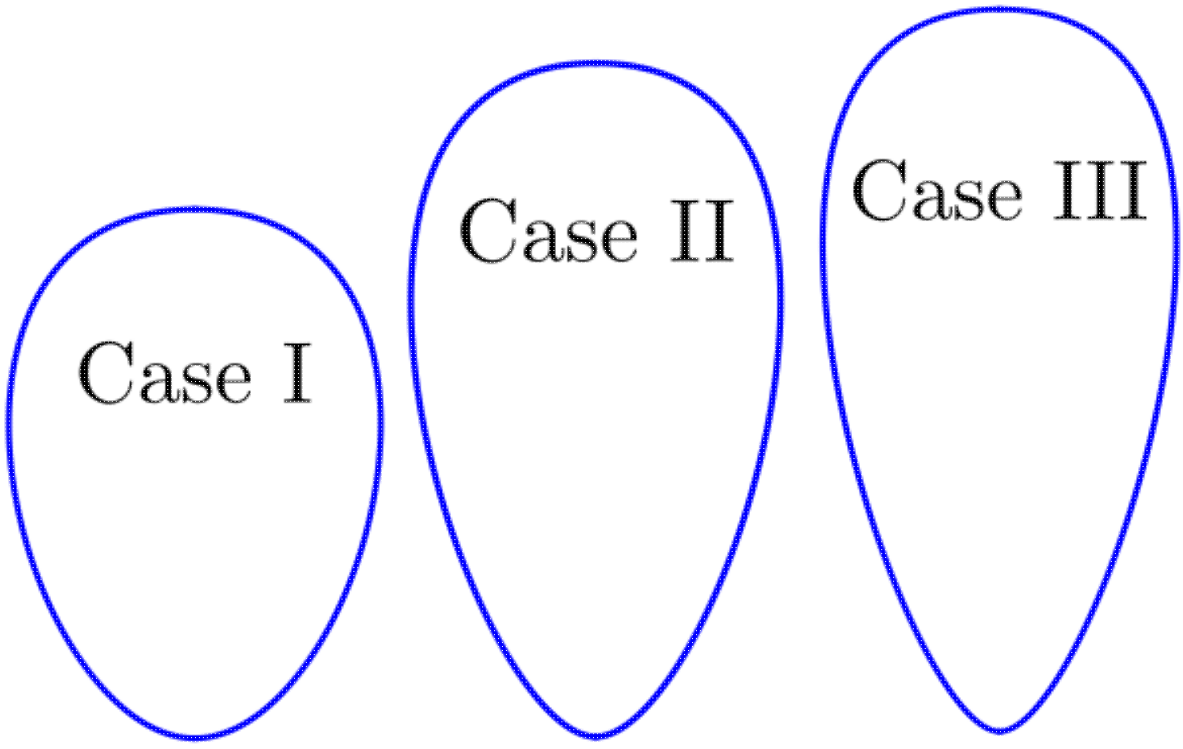
Examples:

```
weld_shape_teardrop width 100.0 case I

weld_shape_teardrop width 100.0 case II

weld_shape_teardrop width 100.0 case III
```

The three examples above are depicted in the images below. These images illustrate cases I, II and III for a fixed width of 100 sites. Because the width is fixed, each case produces a different pool length. NOTE: due to scaling for this documentation, constant *width=100* for each pool shape is not perfectly rendered in images below although they are close.



Description:

Specify size and shape of weld pool at top surface of weld. Shape of pool at root surface (bottom) is controlled by *alpha* in potts/weld. The aspect ratio *length/width* for cases I, II, and III are 1.4, 1.8 and 2.2 respectively. If a

specific width is desired then that is specified directly in the command and pool length is implied by the aspect ratio. On the other hand, if a specific length desired, then the input width must be calculated by hand using the desired length and aspect ratio.

Restrictions:

This command is only valid when used with potts/weld.

Related commands:

weld_shape_ellipse

Default: none