



# Table of Contents

|  |    |
|--|----|
| ChemCell Documentation.....                  | 1  |
| 1. Introduction.....                         | 2  |
| 1.1 What is ChemCell.....                    | 2  |
| 1.2 Open source distribution.....            | 3  |
| 1.3 Acknowledgements.....                    | 3  |
| 2. Getting Started.....                      | 5  |
| 2.1 What's in the ChemCell distribution..... | 5  |
| 2.2 Making ChemCell.....                     | 5  |
| 2.3 Running ChemCell.....                    | 7  |
| 2.4 Command-line options.....                | 8  |
| 2.5 ChemCell screen output.....              | 9  |
| 3. Commands.....                             | 10 |
| 3.1 ChemCell input script.....               | 10 |
| 3.2 Parsing rules.....                       | 11 |
| 3.3 Input script structure.....              | 11 |
| 3.4 Commands listed by category.....         | 12 |
| 3.5 Individual commands.....                 | 13 |
| 4. Example problems.....                     | 14 |
| 5. Additional tools.....                     | 15 |
| 6. Modifying & extending ChemCell.....       | 16 |
| Region geometry options.....                 | 17 |
| Fix options.....                             | 17 |
| Simulator options.....                       | 18 |
| New Top-level Commands.....                  | 18 |
| 7. Errors.....                               | 19 |
| 7.1 Common problems.....                     | 19 |
| 7.2 Reporting bugs.....                      | 20 |
| 7.3 Error & warning Messages.....            | 20 |
| Errors:.....                                 | 20 |
| Warnings:.....                               | 27 |
| balance command.....                         | 28 |
| bin command.....                             | 29 |
| boundary command.....                        | 31 |
| check command.....                           | 32 |
| clear command.....                           | 33 |
| count command.....                           | 34 |
| debug command.....                           | 35 |
| diffusion command.....                       | 36 |
| dimension command.....                       | 37 |
| dump command.....                            | 38 |
| dump_modify command.....                     | 39 |
| echo command.....                            | 40 |
| fix command.....                             | 41 |
| fix conc/random command.....                 | 42 |
| fix conc/set command.....                    | 43 |
| fix dna/toggle command.....                  | 44 |
| fix rate/saturate command.....               | 46 |
| global command.....                          | 47 |

# Table of Contents

|                            |    |
|----------------------------|----|
| include command.....       | 48 |
| jump command.....          | 49 |
| label command.....         | 51 |
| log command.....           | 52 |
| move_style command.....    | 53 |
| move_test command.....     | 54 |
| next command.....          | 55 |
| particles command.....     | 57 |
| permeable command.....     | 58 |
| print command.....         | 59 |
| probability command.....   | 60 |
| react_modify command.....  | 61 |
| reaction command.....      | 64 |
| read_restart command.....  | 65 |
| region command.....        | 66 |
| restart command.....       | 67 |
| run command.....           | 68 |
| run_style command.....     | 69 |
| seed command.....          | 71 |
| shell command.....         | 72 |
| species command.....       | 74 |
| stats command.....         | 75 |
| stats_modify command.....  | 76 |
| timestep command.....      | 77 |
| triangles command.....     | 78 |
| undump command.....        | 79 |
| unfix command.....         | 80 |
| unreact command.....       | 81 |
| variable command.....      | 82 |
| volume command.....        | 86 |
| write_restart command..... | 87 |

# ChemCell Documentation

(10 Sept 2008 version of ChemCell)

ChemCell is a particle-based reaction/diffusion simulator designed for modeling protein networks in biological cells with spatial information. Compartments within the cell are bounded by membrane surfaces defined as geometric regions or connected triangles. Particles represent individual biomolecules and diffuse via Brownian motion in 3d and on 2d surfaces. Particles react with neighboring particles according to rules derived from chemical rate equations. Non-spatial models can also be run using an ODE solver or the Gillespie Stochastic Simulation Algorithm (SSA). Spatial simulations can be run in parallel as can multiple instances of non-spatial simulations for statistical averaging. The [Pizza.py package](#) of Python-based tools can create input files for ChemCell and visualize and plot its output.

ChemCell was developed at Sandia National Laboratories, a US Department of Energy facility, with funding from the DOE. It is an open-source code, designed to be easy to modify and extend with new functionality. It is distributed freely under the terms of the GNU Public License (GPL).

The authors of the code are [Steve Plimpton](#) and Alex Slepoy, who can be contacted at [sjplimp@sandia.gov](mailto:sjplimp@sandia.gov) and [alexander.slepoy@nnsa.doe.gov](mailto:alexander.slepoy@nnsa.doe.gov). The [ChemCell WWW Site](http://www.cs.sandia.gov/~sjplimp/chemcell.html) at [www.cs.sandia.gov/~sjplimp/chemcell.html](http://www.cs.sandia.gov/~sjplimp/chemcell.html) has more information about the code and its uses.

---

The ChemCell 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 us mail so we can improve the ChemCell documentation.

[PDF file](#) of the entire manual, generated by [htmldoc](#)

1. [Introduction](#)
  - 1.1 [What is ChemCell](#)
  - 1.2 [Open source distribution](#)
  - 1.3 [Acknowledgements](#)
2. [Getting started](#)
  - 2.1 [What's in the ChemCell distribution](#)
  - 2.2 [Making ChemCell](#)
  - 2.3 [Running ChemCell](#)
  - 2.4 [Command-line options](#)
  - 2.5 [Screen output](#)
3. [Commands](#)
  - 3.1 [ChemCell input script](#)
  - 3.2 [Parsing rules](#)
  - 3.3 [Input script structure](#)
  - 3.4 [Commands listed by category](#)
  - 3.5 [Commands listed alphabetically](#)
4. [Example problems](#)
5. [Additional tools](#)
6. [Modifying & Extending ChemCell](#)
7. [Errors](#)
  - 7.1 [Common problems](#)
  - 7.2 [Reporting bugs](#)
  - 7.3 [Error & warning messages](#)

## 1. Introduction

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

### [1.1 What is ChemCell](#)

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

### [1.3 Acknowledgments](#)

#### 1.1 What is ChemCell

ChemCell is a particle-based reaction/diffusion simulator designed to model signaling, regulatory, or metabolic networks in biological cells. It can be run as a spatial simulator where the particles diffuse within the specified geometry of a cell, or as a non-spatial simulator where there is no diffusion and the network of chemical reactions is time-integrated as a set of ODEs or stochastically via the Gillespie Stochastic Simulation Algorithm (SSA).

For spatial simulations, a single protein, protein complex, or other biomolecule is represented as a dimensionless particle. The geometry of a cell (membranes, organelles, etc) is represented as simple geometrical objects (spheres, boxes, etc) or as triangulated surfaces. Particles diffuse randomly via Brownian motion, either in 3d or on 2d surfaces. Biochemical reactions occur in accord with chemical rate equations, which are inputs to ChemCell. Monte Carlo rules are used to perform reactions each timestep between pairs of nearby particles.

For non-spatial simulations, the cell is treated as a well-mixed chemical reactor, and the chemical rate equations can be time-integrated either by a fixed-step or adaptive-timestep ODE solver or by the direct-method variant of the Gillespie Stochastic Simulation Algorithm (SSA) using the Gibson/Bruck methodology.

ChemCell runs on single-processor desktop or laptop machines, but can also be run in parallel. Multiple runs can be performed on a collection of processors (for statistical purposes), or (for spatial simulations) the simulation domain can be partitioned across processors and a single simulation run in parallel. For spatial simulations, ChemCell can model systems with only a few particles up to many millions.

ChemCell is a freely-available open-source code, distributed under the terms of the [GNU Public License](#), which means you can use or modify the code however you wish. See [this section](#) for a brief discussion of the open-source philosophy.

ChemCell is designed to be easy to modify or extend with new capabilities, such as modified reaction rules or boundary conditions. See [this section](#) for more details.

ChemCell is written in C++ and can be downloaded from the [ChemCell WWW Site](#). No additional software is needed to run ChemCell in serial on a desktop machine. ChemCell will run on any parallel machine that compiles C++ and supports the [MPI](#) message-passing library. This includes distributed- or shared-memory parallel machines and Beowulf-style clusters. To run with spatial parallelism, the [Zoltan library](#) must also be installed.

ChemCell does not have the ability to create cellular geometries or visualize simulation output. Currently, these tasks are handled by pre- and post-processing codes. Our group has written a separate toolkit called [Pizza.py](#) which provides tools for doing setup, analysis, plotting, and visualization for ChemCell simulations. Pizza.py is written in [Python](#) and is available for download from [the Pizza.py WWW site](#).

These are other cell-modeling simulation codes. The first 3 are similar in spirit to ChemCell, meaning they model biochemical networks of reacting/diffusing chemical species within a spatial geometry via particles. VCell uses

continuum PDEs for reaction/diffusion in a spatial representation of the cell. E-Cell provides a rich set of continuum and stochastic solvers for non-spatial cell models.

- [MCell](#) – by Joel Stiles (PSC) and Tom Bartol (Salk Institute)
  - [MesoRD](#) – by Johan Hattne (EMBL) and Johan Elf (Uppsala Univ)
  - [Smoldyn](#) – by Steve Andrews (TMSI)
  
  - [VCell](#) – by Jim Schaaf and Les Loew (NRCAM, U Conn Health Center)
  - [E-Cell](#) – a large Japanese-led project
- 

## 1.2 Open source distribution

ChemCell comes with no warranty of any kind. As each source file states in its header, it is a copyrighted code that is distributed free-of-charge, under the terms of the [GNU Public License](#) (GPL). This is often referred to as open-source distribution – see [www.gnu.org](http://www.gnu.org) or [www.opensource.org](http://www.opensource.org) for more details. The legal text of the GPL is in the LICENSE file that is included in the ChemCell distribution.

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

- (1) Anyone is free to use, modify, or extend ChemCell in any way they choose, including for commercial purposes.
- (2) If you distribute a modified version of ChemCell, it must remain open-source, meaning you distribute it under the terms of the GPL. You should clearly annotate such a code as a derivative version of ChemCell.
- (3) If you release any code that includes ChemCell source code, then it must also be open-sourced, meaning you distribute it under the terms of the GPL.
- (4) If you give ChemCell 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 ChemCell for something useful or if you fix a bug or add a new feature or application to the code, let us know. We would like to include your contribution in the released version of the code and/or advertise your success on our WWW page.

---

## 1.3 Acknowledgements

ChemCell is distributed by [Sandia National Laboratories](#). ChemCell development has been funded by the [US Department of Energy](#) (DOE), through its LDRD program and through its [Genomes-to-Life program](#) via the [ASCR](#) and [BER](#) offices.

The original version of ChemCell was part of the [GTL project](#) "Carbon Sequestration in Synechococcus Sp.: From Molecular Machines to Hierarchical Modeling".

If you use ChemCell results in your published work, please provide a link to the [ChemCell WWW page](#) and cite one of the ChemCell papers listed there.

If you send information about your publication, we'll be pleased to add it to the Publications page of the [ChemCell WWW Site](#). Ditto for a picture or movie for the Pictures or Movies pages.

The authors of ChemCell are [Steve Plimpton](#) and Alex Slepoy of Sandia National Labs who can be contacted via email at [sjplimp@sandia.gov](mailto:sjplimp@sandia.gov) or [alexander.slepoy@nnsa.doe.gov](mailto:alexander.slepoy@nnsa.doe.gov).

We thank Larry Lok and Roger Brent at the Molecular Sciences Institute (TMSI) for help in designing ChemCell. We also thank Dan Gillespie for constructive feedback on algorithms and Steve Andrews (LBNL, now TMSI) for helping us understand and use output from his Smoldyn code in ChemCell.

## 2. Getting Started

This section describes how to unpack, make, and run ChemCell, for both new and experienced users.

[2.1 What's in the ChemCell distribution](#) [2.2 Making ChemCell](#) [2.3 Running ChemCell](#) [2.4 Command–line options](#)  
[2.5 Screen output](#)

---

### 2.1 What's in the ChemCell distribution

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

```
gunzip ChemCell*.tar.gz
tar xvf ChemCell*.tar
```

This will create a ChemCell directory containing two files and several sub–directories:

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

---

### 2.2 Making ChemCell

*Read this first:*

Building ChemCell can be non–trivial. You will likely need to edit a makefile, there are compiler options, additional libraries can be used (MPI, Zoltan), 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 ChemCell executable:*

The src directory contains the C++ source and header files for ChemCell. It also contains a top–level Makefile and a MAKE directory with low–level Makefile.\* files for several machines. From within the src directory, type "make" or "gmake". You should see a list of available choices. If one of those is the machine and options you want, you can type a command like:

```
make linux
gmake mac
```

Note that on a multi–processor or multi–core platform you can launch a parallel make, by using the "–j" switch with the make command, which will typically build ChemCell more quickly.

If you get no errors and an executable like spk\_linux or spk\_mac is produced, you're done; it's your lucky day.

*Errors that occur when making ChemCell:*



(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 directory. Use whatever existing file is closest to your platform as a starting point. See the next section for more instructions.

### ***Editing a new low-level Makefile.foo:***

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

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

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

(3) If you want ChemCell to run in parallel, you must have two libraries installed on your platform: MPI and Zoltan. For MPI, Makefile.foo needs to specify where the mpi.h file (-I switch) and the libmpi.a library (-L switch) is found. If you are installing MPI yourself, we recommend Argonne's MPICH 1.2 or 2.0 which can be downloaded from the [Argonne MPI site](#). OpenMPI should also work. If you are running on a big parallel platform, your system people or the vendor should have already installed a version of MPI, which will be faster than MPICH or OpenMPI, so find out how to build and link with it. If you use MPICH or OpenMPI, you will have to configure and build it for your platform. The MPI configure script should have compiler options to enable you to use the same compiler you are using for the ChemCell build, which can avoid problems that may arise when linking ChemCell to the MPI library.

Zoltan is an open-source parallel load-balancing library, also distributed by Sandia National Labs. It can be downloaded at [this site](#). Follow its installation instructions. It builds out-of-the-box for many machines. If not for yours, you will need to edit a zoltan/Utilities/Config/Config.\* file suitable for your platform. Once a Zoltan library exists on your machine, add the appropriate -I, -L, and -l switches to your Makefile.foo using one of the other MAKE/Makefile.\* files as a template. Note that there are 3 Zoltan libraries you need to link to: zoltan, zoltan\_mem, and zoltan\_comm.

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

The file STUBS/mpi.cpp has a CPU timer function MPI\_Wtime() that calls gettimeofday(). If your system doesn't support gettimeofday(), you'll need to insert code to call another timer. Note that the ANSI-standard function clock() rolls over after an hour or so, and is therefore insufficient for timing long ChemCell runs.

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

That's it. Once you have a correct Makefile.foo and you have pre-built the MPI and Zoltan libraries it will use, 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 ccell\_foo when the build is complete.

### ***Additional build tips:***

(1) Building ChemCell for multiple platforms.

You can make ChemCell for multiple platforms from the same src directory. Each target creates its own object sub-dir 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 ChemCell is built.

(3) Building for a Macintosh.

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

(4) Building for MicroSoft Windows.

I've never done this, but ChemCell is just standard C++ with MPI and Zoltan calls. You should be able to use cygwin to build ChemCell with a Unix-style make. Or you should be able to pull all the source files into Visual C++ (ugh) or some similar development environment and build it. Good luck – I can't help you on this one.

---

## **2.3 Running ChemCell**

By default, ChemCell runs by reading commands from stdin; e.g. ccell\_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 ChemCell on any of the sample inputs provided in the examples directory. Input scripts are named in.\* and sample outputs are named log.\*.

Here is how you might run the simple A + B C reaction network on a Linux box.

```
cd src
make linux
cp ccell_linux ../examples/abc
cd ../examples/abc
ccell_linux <in.abc
```

If you wanted to run in parallel, mpirun could be used to launch ChemCell, replaing the last command with

```
mpirun -np 4 ccell_linux <in.abc
```

The screen output from ChemCell is described in the next section. As it runs, ChemCell also writes a log.ccell file with the same information. Note that this sequence of commands copied the ChemCell executable (ccell\_linux) to the directory with the input files. If you don't do this, ChemCell may look for input files or create output files in the directory where the executable is, rather than where you run it from.

If ChemCell 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 ChemCell detects, a list of all ERROR and WARNING messages, and what to do about them.

For spatial simulations ChemCell can run a problem on any number of processors, including a single processor. In principle, you should get identical answers on any number of processors and on any machine. In practice, numerical round-off on different machines can cause slight differences and eventual divergence of two simulations.

ChemCell 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.

---

## 2.4 Command-line options

At run time, ChemCell recognizes several optional command-line switches which may be used in any order. For example, `ccell_ibm` might be launched as follows:

```
mpirun -np 16 ccell_ibm -var f tmp.out -log my.log -screen none <in.ecoli
```

These are the command-line options:

`-partition 8x2 4 5 ...`

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

`-in file`

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

`-log file`

Specify a log file for ChemCell to write status information to. In one-partition mode, if the switch is not used, ChemCell writes to the file `log.ccell`. If this switch is used, ChemCell writes to the specified file. In multi-partition mode, if the switch is not used, a `log.ccell` file is created with hi-level status information. Each partition also writes to a `log.ccell.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 ChemCell to write its screen information to. In one-partition mode, if the switch is not used, ChemCell writes to the screen. If this switch is used, ChemCell 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 X value

Specify a variable that will be defined for substitution purposes when the input script is read. X should be a single lower-case character from 'a' to 'z'. The value can be any string. Using this command-line option is equivalent to putting the line "variable X index value" at the beginning of the input script. See the [variable](#) command for more information.

---

## 2.5 ChemCell screen output

As ChemCell 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, ChemCell performs various initializations and prints information about species, diffusion, reactions, and binning (used to find nearby particles). It also prints details of the initial species counts for the system. During the run itself, species counts are printed periodically, every few timesteps. When the run concludes, ChemCell prints the final species counts and a total run time for the simulation. It then appends additional statistics about the run. An example set of statistics is shown here:

Loop time of 14.0014 on 1 procs for 100 steps

Move time (%) = 0.9302 (6.6436) Migrt time (%) = 0.809228 (5.7796) React time (%) = 12.2261 (87.32) RComm time (%) = 0.0172122 (0.122931) Outpt time (%) = 0.0185347 (0.132377) Balnc time (%) = 0 (0) Other time (%) = 0.000203848 (0.00145591)

Nlocal: 3641 ave 3641 max 3641 min Histogram: 1 0 0 0 0 0 0 0 0 Nghost: 2083 ave 2083 max 2083 min Histogram: 1 0 0 0 0 0 0 0 0 Nbin: 512 ave 512 max 512 min Histogram: 1 0 0 0 0 0 0 0 0

Move statistics (total &per-step): moves = 379615 3796.15 tri checks = 0 0 refl hits = 0 0 near hits = 0 0 stick hits = 0 0 far hits = 0 0 thru hits = 0 0 Reaction statistics (total &per-step): bin-bin = 302400 3024 bin pairs = 90104885 901049 dist checks = 27858171 278582 overlaps = 914444 9144.44 reactions = 397 3.97 count of each reaction: (1 378) (2 19) Number of balancing calls = 0 Memory usage in Mbyte/proc (ave/max/min) parts = 0.34758 0.34758 0.34758 bins = 0.288513 0.288513 0.288513 surfs = 0 0 0 total = 0.636093 0.636093 0.636093 Equivalence map of species &type map A 1 map B 2 map C 3

The first section gives the breakdown of the CPU run time (in seconds) into major categories. The second section lists the number of owned particles (Nlocal), ghost particles (Nghost), and bins stored by processor. The max and min values give the spread of these values across processors with a 10-bin histogram showing the distribution. The total number of histogram counts is equal to the number of processors.

The last section gives aggregate statistics for diffusion and reactions during the run. The memory usage per processor is summarized. And a mapping of species names to index numbers is given which is useful for analyzing dump files of particle coordinates.

## 3. Commands

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

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

### 3.1 ChemCell input script

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

```
stats      10
run        100
run        100
```

does something different than this sequence:

```
run        100
stats      10
run        100
```

In the first case, statistics are printed every 10 timesteps during two simulations of 100 timesteps each. In the 2nd case, the default statistics setting (0) is used for the 1st 100 step simulation and the setting of 10 is used for the 2nd one. Thus statistics during the 1st run will only be printed at the beginning and end of the run.

(2) Some commands are only valid when they follow other commands. For example you cannot setup a reaction until the particle species it specifies have been defined.

(3) Sometimes command B will use values that can be set by command A. This means command A must precede command B in the input script if it is to have the desired effect. For example, the [particles](#) command reads a group of particles and assigns them random numbers using a random number generator. If default values are not desired, the [seed](#) command must be used before particles to tell ChemCell how to initialize the random number generator.

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

- (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 `$` characters. If the character following the `$` is "a" to "z", the two-character sequence (e.g. `$x`) is replaced with the corresponding variable text. See the [variable](#) command for details.
  - (4) The line is broken into "words" separated by whitespace (tabs, spaces). Note that words can thus contain letters, digits, underscores, or punctuation characters.
  - (5) The first word is the command name. All successive words in the line are arguments.
  - (6) Text with spaces can be enclosed in double quotes so it will be treated as a single argument. See the [fix print](#) command for an example. A `#` or `$` character in text between double quotes will also 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 ChemCell input script. The "examples" directory in the ChemCell distribution contains many sample input scripts; the corresponding problems are discussed in [this section](#), and animated on the [ChemCell WWW Site](#).

A ChemCell input script typically has several stages:

1. Initialization
2. Define bins
3. Define geometry
4. Define particles
5. Define reactions
6. Settings
7. Run a simulation

The last 2 stages can be repeated as many times as desired. I.e. run a simulation, change some settings, run some more, etc. The "Define bins" and "Define geometry" stages are not used for non-spatial simulations (ODE or Gillespie). Each of the stages is now described in more detail. Remember that almost all the commands need only be used if a non-default value is desired.

### (1) Initialization

Set parameters that are typically defined before the simulation domain is binned.

The relevant commands are [run\\_style](#), [global](#), [volume](#), [boundary](#), [seed](#), [timestep](#), [move\\_style](#).

## (2) Define bins

For spatial simulations, the simulation domain is partitioned into 3d bins using the [bin](#) command. This must be done before geometric objects or particles are created or read-in.

## (3) Define geometry

The [triangles](#) and [region](#) commands are used to create geometric objects and membrane boundaries.

## (4) Define particles

Particles and their attributes are created and set via these commands:

[species](#), [dimension](#), [diffusion](#), [count](#), [permeable](#), [particles](#), [read\\_restart](#)

## (5) Define reactions

Reactions (zero-order, unary, binary) are defined via the [reaction](#), [react\\_modify](#), and [probability](#) commands.

## (6) Settings

Before a simulation is run, a variety of settings can be specified using these commands:

[balance](#), [dump](#), [stats](#), [fix](#), [restart](#)

## (7) Run a simulation

A spatial or non-spatial simulation is run using the [run](#) command.

---

### 3.4 Commands listed by category

This section lists all ChemCell commands, grouped by category. The [next section](#) lists the same commands alphabetically.

Settings:

[run\\_style](#), [move\\_style](#), [seed](#), [balance](#), [timestep](#)

Geometry:

[boundary](#), [global](#), [region](#), [triangles](#), [volume](#)

Particles:

[species](#), [count](#), [diffusion](#), [dimension](#), [permeable](#), [particles](#), [read\\_restart](#)

Reactions:

[reaction](#), [react\\_modify](#), [unreact](#), [probability](#)

Fixes:

[fix](#), [unfix](#)

Output:

[dump](#), [restart](#), [stats](#), [stats\\_modify](#), [dump\\_modify](#), [undump](#), [write\\_restart](#), [check](#), [debug](#)

Actions:

[bin](#), [move\\_test](#), [run](#),

Miscellaneous:

[clear](#), [echo](#), [include](#), [jump](#), [label">log](#), [next](#), [print](#), [shell](#), [variable](#)

---

### 3.5 Individual commands

This section lists all ChemCell commands alphabetically. The [previous section](#) lists the same commands, grouped by category.

|                              |                              |                               |                              |                             |                             |
|------------------------------|------------------------------|-------------------------------|------------------------------|-----------------------------|-----------------------------|
| <a href="#">balance</a>      | <a href="#">bin</a>          | <a href="#">boundary</a>      | <a href="#">check</a>        | <a href="#">clear</a>       | <a href="#">count</a>       |
| <a href="#">debug</a>        | <a href="#">diffusion</a>    | <a href="#">dimension</a>     | <a href="#">dump</a>         | <a href="#">dump_modify</a> | <a href="#">echo</a>        |
| <a href="#">fix</a>          | <a href="#">global</a>       | <a href="#">include</a>       | <a href="#">jump</a>         | <a href="#">label</a>       | <a href="#">log</a>         |
| <a href="#">move_style</a>   | <a href="#">move_test</a>    | <a href="#">next</a>          | <a href="#">particles</a>    | <a href="#">permeable</a>   | <a href="#">probability</a> |
| <a href="#">print</a>        | <a href="#">react_modify</a> | <a href="#">reaction</a>      | <a href="#">read_restart</a> | <a href="#">region</a>      | <a href="#">restart</a>     |
| <a href="#">run</a>          | <a href="#">run_style</a>    | <a href="#">seed</a>          | <a href="#">shell</a>        | <a href="#">species</a>     | <a href="#">stats</a>       |
| <a href="#">stats_modify</a> | <a href="#">timestep</a>     | <a href="#">triangles</a>     | <a href="#">undump</a>       | <a href="#">unfix</a>       | <a href="#">unreact</a>     |
| <a href="#">variable</a>     | <a href="#">volume</a>       | <a href="#">write_restart</a> |                              |                             |                             |

---

Fix styles. See the [fix](#) command for one-line descriptions of each style or click on the style itself for a full description:

|                             |                          |                            |                               |
|-----------------------------|--------------------------|----------------------------|-------------------------------|
| <a href="#">conc/random</a> | <a href="#">conc/set</a> | <a href="#">dna/toggle</a> | <a href="#">rate/saturate</a> |
|-----------------------------|--------------------------|----------------------------|-------------------------------|



## 4. Example problems

The ChemCell distribution includes an examples sub-directory with several sample problems. Each problem is in a sub-directory of its own. Each problem has an input script (in.\*) and produces a log file (log.\*). It may also produce a dump file (dump.\*) when it runs. Some use one or more data files (data.\*) of geometry or particle info as additional input. These were created via the [Pizza.py](#) pre-processing tool using the \*.py scripts in the same directory.

Plots can be made from the log files using the "clog" and plotting tools in Pizza.py. Images and movies can be made from the dump and data files using the "cdata", "dump", and various visualization tools in Pizza.py. Examples are shown on the [ChemCell WWW Site](#).

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

|        |   |
|--------|---|
| abc    | simple A + B → C reaction                 |
| kinase | 3-stage kinase cascade                    |
| lotka  | Lotka–Volterra predator–prey system       |
| ecoli  | E Coli chemotaxis signaling pathway model |

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

```
cd kinase
cp ../../src/ccell_linux .      # copy ChemCell executable to this dir
ccell_linux <in.kinase          # run the problem
```

Running the simulation produces the files *dump.kinase* and *log.ccell*. Assuming you have Pizza.py installed, you could visualize the dump file as follows:

```
pizza -f dview.py dump.kinase
```

## 5. Additional tools

ChemCell is designed to be a computational kernel for performing reaction/diffusion computations. Additional pre- and post-processing steps are typically needed to setup and analyze a simulation.

Our group has written and released a separate toolkit called [Pizza.py](#) which provides tools for doing setup, analysis, plotting, and visualization for ChemCell simulations. Pizza.py is written in [Python](#) and is available for download from [the Pizza.py WWW site](#). Images and movies created by Pizza.py are displayed on the [ChemCell WWW site](#).

Pizza.py has a "cdata" tool with many sub-commands for reading and creating regions, surfaces, particles. The resulting geometries can be visualized with other Pizza.py tools: "gl", "vcr", "raster", "svg", etc.

Similarly, Pizza.py has a "dump" tool which reads in ChemCell dump files and allows particles to be selected, visualized, and animated, along with the ChemCell geometry (regions, triangulated surfaces) via the same Pizza.py tools.

Pizza.py also has a "clog" tool which reads in ChemCell log files and allows the species concentration statistics to be plotted either via GnuPlot ("gnu" tool) or MatLab ("matlab" tool).

If you write additional tools (either stand-alone or as part of Pizza.py) that you think are generally useful for setting up or analyzing ChemCell simulations, send us an email; we can include them as part of the ChemCell or Pizza.py packages.

## 6. Modifying & extending ChemCell

ChemCell 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 some minimal instructions. Realistically, the best way to add a new feature is to find a similar feature in ChemCell 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 ChemCell and its class organization, but functions (class methods) that do actual computations are written in vanilla C-style code and typically operate on simple C-style data structures (vectors and arrays).

The new features described in this section require you to write a new C++ class. Creating a new class requires 2 files, a source code file (\*.cpp) and a header file (\*.h). Their contents are briefly discussed below. Enabling ChemCell to invoke the new class is as simple as adding two definition lines to the style.h file, in the same syntax as the existing ChemCell classes are defined in the style.h file.

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

Here is a concrete example. Suppose you write 2 files fix\_foo.cpp and fix\_foo.h that define a new class FixFoo that implements a boundary condition described in the classic 1997 [paper](#) by Foo, et. al. If you wish to invoke that boundary condition in a ChemCell input script with a command like

```
fix 1 foo 0.1 3.5
```

you simply need to put your 2 files in the ChemCell src directory, add 2 lines to the style.h file, and re-make the code.

The first line added to style.h would be

```
FixStyle(foo,FixFoo)
```

in the #ifdef FixClass section, where "foo" is the style keyword in the pair\_style command, and FixFoo is the class name in your C++ files.

The 2nd line added to style.h would be

```
#include "fix_foo.h"
```

in the #ifdef FixInclude section, where fix\_foo.h is the name of your new include file.

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

Note that if you are using Makefile.list instead of Makefile to build ChemCell, you will need to explicitly add the names of your new .cpp and .h file to Makefile.list.

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

- [Region geometry options](#)
- [Fix options](#) which include boundary conditions, diagnostic output, etc
- [Simulator options](#)
- [New top-level commands](#)

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

The instructions below for each category will list the header file for the parent class that these styles are sub-classes of. Public variables in that file are ones used and set by the sub-classes which are also used by the parent class. Sometimes they are also used by the rest of ChemCell. Virtual functions in the header file which are set = 0 are ones you must define in your new class to give it the functionality ChemCell expects. Virtual functions that are not set to 0 are functions you can optionally define.

---



---

## Region geometry options

Classes that define simple geometric regions are sub-classes of the Region class. See the region.h file for a list of methods these classes defines. Regions are used in ChemCell to define surfaces which particles diffuse between or on.

Region\_sphere.cpp and region\_sphere.h are the simplest example of a Region class. They implement the *sphere* style of the [region](#) command.

Here is a brief description of the class methods required:

|                |   |
|----------------|---|
| bbox           | bounding box                                    |
| inside         | determine whether a point is in the region      |
| hex_intersect  | check if region surface intersects a hex cell   |
| line_intersect | check if line segment intersects region surface |
| compute_normal | unit normal to region at a point                |
| move2d         | perform a 2d move on region surface             |
| distance       | compute distance from point to region surface   |

---

## Fix options

In ChemCell, a "fix" is any operation that is computed during timestepping that alters some property of the system. It could be alteration of a rate or a rate equation, implementation of a boundary condition, or calculation of some diagnostic property. See the fix.h file for a list of methods these classes defines.

Here is a brief description of the class methods. All of these methods are optional.

|         |  |
|---------|--|
| init    | called before timestepping               |
| initial | called at the beginning of each timestep |
| final   | called at the end of each timestep       |

|         |                           |
|---------|---------------------------|
| cleanup | called after timestepping |
|---------|---------------------------|

---

## Simulator options

All classes that define an atom style are sub-classes of the Atom class. See the atom.h file for a list of methods these classes defines. The atom style determines what quantities are associated with an atom in a ChemCell simulation. If one of the existing atom styles does not define all the arrays you need to store with an atom, then a new atom class can be created.

Atom\_atomic.cpp and atom\_atomic.h are the simplest example of an Atom class. They implement the *atomic* style of the [atom\\_style](#) command.

Here is a brief description of the class methods required:

|      |                    |
|------|--------------------|
| init | setup a simulation |
| run  | the timestepper    |

---

## New Top-level Commands

It is possible to add a new command to a ChemCell input script as opposed to adding a new style to an existing command (region, fix, simulator). For example the run and write\_restart commands are top-level ChemCell commands that are listed in the Command section of style.h. When such a command is encountered in the ChemCell input script, the topmost level of ChemCell (ChemCell.cpp) 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 the ChemCell data structures.

Thus to add a new command, you simply need to add a \*.cpp and \*.h file containing a single class:

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

Of course, the new class can define other methods and variables that it uses internally.

---

(**Foo**) Foo, Morefoo, and Maxfoo, J of Biological Boundary Conditions, 75, 345 (1997).

## 7. Errors

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

[7.1 Common problems](#)

[7.2 Reporting bugs](#)

[7.3 Error & warning messages](#)

---

### 7.1 Common problems

If two ChemCell runs do not produce the same answer on different machines or different numbers of processors, this is typically not a bug. In theory you should get identical answers on any number of processors and on any machine. In practice, numerical round-off can cause slight differences and eventual divergence of molecular dynamics phase space trajectories within a few 100s or few 1000s of timesteps. However, the statistical properties of the two runs (e.g. average energy or temperature) should still be the same.

If the [velocity](#) command is used to set initial atom velocities, a particular atom can be assigned a different velocity when the problem is run on different machines. Obviously, this means the phase space trajectories of the two simulations will rapidly diverge. See the discussion of the *loop* option in the [velocity](#) command for details.

A ChemCell simulation typically has two stages, setup and run. Most ChemCell errors are detected at setup time; others like a bond stretching too far may not occur until the middle of a run.

ChemCell tries to flag errors and print informative error messages so you can fix the problem. Of course ChemCell cannot figure out your physics mistakes, like choosing too big a timestep, specifying invalid force field coefficients, or putting 2 atoms on top of each other! If you find errors that ChemCell doesn't catch that you think it should flag, please send us an [email](#).

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.ChemCell file or using the [echo command](#) to see it on the screen. For a given command, ChemCell expects certain arguments in a specified order. If you mess this up, ChemCell will often flag the error, but it may read a bogus argument and assign a value that is valid, but not what you wanted. E.g. trying to read the string "abc" as an integer value and assigning the associated variable a value of 0.

Generally, ChemCell 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 ChemCell 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:

ChemCell runs in the available memory a processor allows to be allocated. Most reasonable MD runs are compute limited, not memory limited, so this shouldn't be a bottleneck on most platforms. Almost all large memory allocations in the code are done via C-style malloc's which will generate an error message if you run out of memory. Smaller chunks of memory are allocated via C++ "new" statements. If you are unlucky you could run out of memory just when one of these small requests is made, in which case the code will crash or hang (in parallel), since ChemCell doesn't trap on those errors.

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

In parallel, one way ChemCell 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.

---

## 7.2 Reporting bugs

If you are confident that you have found a bug in ChemCell, we'd like to know about it via [email](#).

First, check the "New features and bug fixes" section of the [ChemCell 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 number of atoms and fewest number of processors and with the simplest input script that reproduces the bug.

Send an email that describes 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.

---

## 7.3 Error &warning Messages

These are two alphabetic lists of the [ERROR](#) and [WARNING](#) messages ChemCell 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:

*%d local particles on proc %d are in ghost cells after compact*

Debug test to insure particles have moved to correct processors. This error should not occur.

*%d particles assigned to procs*

Particles read from file, but not all assigned correctly to processors.

*2d particle move hit a triangle vertex*

Particle moving on a surface went exactly through a triangle vertex. Should not occur.

*Alias %s already exists*

Cannot reuse a species name.

*All universe/uloop variables must have same # of values*

*All variables in next command must be same style*

Self-explanatory.

*Another input script is already being processed*

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

*Arccos of invalid value in variable formula*

Argument of arccos() must be between -1 and 1.

*Arcsin of invalid value in variable formula*

Argument of arcsin() must be between -1 and 1.

*Bin size of 0.0*

Self-explanatory.

*Bins are already setup*

Global and boundary command must be used before bins command. Bins command can only be used once.

*Cannot add triangles to a region surface*

Triangles can only be added to a triangulated surface.

*Cannot change dimension of existing particles*

Particles are either 2d or 3d. Cannot be changed.

*Cannot check for a 2d species*  
 Check command for inside/outside a surface can only be done for 3d species.

*Cannot do move\_test with timestep = 0*  
 Timestep must be non-zero to perform a move test.

*Cannot have 0-reactant reactions with spatial simulation*  
 This is because the volume within which to create the particles is not well-defined.

*Cannot open fix conc/random file %s*  
 Self-explanatory.

*Cannot open fix conc/set file %s*  
 Self-explanatory.

*Cannot open input script %s*  
 Self-explanatory.

*Cannot open logfile %s*  
 The ChemCell log file specified in the input script cannot be opened. Check that the path and name are correct.

*Cannot open restart file %s*  
 Self-explanatory.

*Cannot perform match with two or more wildcard \**  
 A string with a wildcard character "\*" can only have one wildcard.

*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 set permeability for a 2d species*  
 Permeability only makes sense for a 3d species hitting a surface.

*Cannot use balance command with non-spatial simulation*  
 Self-explanatory.

*Cannot use bin command with non-spatial simulation*  
 Self-explanatory.

*Cannot use count command with spatial simulation*  
 Self-explanatory.

*Cannot use diffusion command with non-spatial simulation*  
 Self-explanatory.

*Cannot use dimension command with non-spatial simulation*  
 Self-explanatory.

*Cannot use dump command with non-spatial simulation*  
 Self-explanatory.

*Cannot use fix conc/random with non-spatial simulations*  
 Self-explanatory.

*Cannot use fix conc/set with spatial simulations*  
 Self-explanatory.

*Cannot use fix dna/toggle with spatial simulations*  
 Self-explanatory.

*Cannot use fix rate/saturate with spatial simulations*  
 Self-explanatory.

*Cannot use global command with non-spatial simulation*  
 Self-explanatory.

*Cannot use move\_style command with non-spatial simulation*  
 Self-explanatory.

*Cannot use move\_test command with non-spatial simulation*  
 Self-explanatory.

*Cannot use particle command with non-spatial simulation*  
 Self-explanatory.



*Cannot use permeable command with non-spatial simulation*  
Self-explanatory.

*Cannot use probability command with non-spatial simulation*  
Self-explanatory.

*Cannot use region command with non-spatial simulation*  
Self-explanatory.

*Cannot use this run style in parallel*  
Only some run styles support parallel execution.

*Cannot use timestep command with gillespie simulation*  
Stochastic simulations set the timestep for each reaction.

*Cannot use triangles command with non-spatial simulation*  
Self-explanatory.

*Cannot use volume command with spatial simulation*  
Self-explanatory.

*Cannot write dump files for non-spatial simulation*  
Self-explanatory.

*Cannot write restart files for non-spatial simulation*  
Restart files only contain particle info, so they are not used for non-spatial simulations.

*Code is not compiled with move debug option*  
Debug command cannot be used unless ChemCell was compiled with `-DDEBUG_MOVE` flag or with the `"#define DEBUG_MOVE"` line uncommented at the top of `src/move.cpp`.

*Could not find reaction ID*  
Self-explanatory.

*Could not find undump ID*  
Self-explanatory.

*Could not find unfix ID*  
Self-explanatory.

*Could not find unreact ID*  
Self-explanatory.

*Could not open balance file*  
The output file used by the balance command could not be opened.

*Could not open dump file*  
The output file used by the dump command could not be opened.

*Could not open input script*  
Self-explanatory.

*Could not open log.ccell*  
Self-explanatory.

*Could not open logfile*  
Self-explanatory.

*Could not open move\_test file*  
The output file used by the move\_test command could not be opened.

*Could not 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.

*Could not 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.

*Could not 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.

*Could not find dump\_modify ID*  
Self-explanatory.

*Did not assign all particles correctly*

Reading of restart file did not assign all particles to processors correctly.

*Did not find matching reaction particle*

After particle communication, matching particle on another processor could not be found. This error should not occur.

*Divide by 0 in variable formula*

Self-explanatory.

*Dump species count does not match particle species count*

Species count has changed since dump file was defined.

*Failed to allocate %d bytes for array %s*

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

*Failed to reallocate %d bytes for array %s*

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

*Global bin does not map to local domain*

Problem with converting global bin ID to a local bin ID. This 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 ChemCell to see the offending line.

*Invalid diffusion constant*

Diffusion coefficient must be  $\geq 0$ .

*Invalid dimensionality*

Dimensionality must be 2 or 3.

*Invalid dump frequency*

Dump frequency must be 1 or greater.

*Invalid fix style*

Self-explanatory.

*Invalid region style*

Self-explanatory.

*Invalid run style*

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 2d particle move*

Problem occurred with 2d particle moving on a triangle. This error should not occur.

*Invalid command-line argument*

One or more command-line arguments is invalid. Check the syntax of the command you are using to launch ChemCell.

*Invalid flag in header of restart file*

Format of restart file was not recognized. This error should not occur unless restart file is corrupted.

*Invalid math function in variable formula*

The math or group function is not recognized.

*Invalid reaction ID in fix dna/toggle command*

Self-explanatory.

*Invalid reaction ID in fix rate/saturate command*

Self-explanatory.

*Invalid region arguments*

Number of arguments doesn't match region style.

*Invalid settings for reaction %s*

One or more settings for a reaction are invalid. This error should not occur.

*Invalid species %s in stats command*  
Self-explanatory.

*Invalid species ID in fix conc/random command*  
Self-explanatory.

*Invalid species ID in fix conc/set command*  
Self-explanatory.

*Invalid species ID in fix dna/toggle command*  
Self-explanatory.

*Invalid species ID in fix rate/saturate command*  
Self-explanatory.

*Invalid syntax in variable formula*  
Self-explanatory.

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

*Invalid variable in next command*  
Self-explanatory.

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

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

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

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

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

*Max 2d move distance > bin size*  
Move distance exceeds bin size. Must increase bin size or reduce move size.

*Max 3d move distance > bin size*  
Move distance exceeds bin size. Must increase bin size or reduce move size.

*Max reaction distance > bin size*  
Binary reaction distance exceeds bin size. Must increase bin size or reduce reaction distance.

*Move to next bin out of range*  
Particle moved into bin not owned by this processor. This error should not occur.

*Must be 2 or more bins in periodic dimensions*  
Binning of global domain has resulted in only one bin in a particular dimension.

*Must print stats in count units for spatial simulation*  
Self-explanatory.

*Must set bins before defining region*  
Use bin command before using region command.

*Must set bins before read restart*  
Use bin command before using read\_restart command.

*Must set bins before reading particles*  
Use bin command before using particles command.

*Must set bins before reading surface*  
Use bin command before using triangles command.

*Must set bins before run*  
Use bin command before using run command.

*Must set global domain before bins*  
Use global command before using bin command.

*Must set run\_style first*

Cannot use this command before the run style is set

*Must set simulation domain via global command*  
Global command has not been used before a simulation is being initialized.

*Must set volume for run style gillespie*  
This run style requires that the volume command be used.

*Must set volume for run style ode/rk*  
This run style requires that the volume command be used.

*Must set volume for run style ode*  
This run style requires that the volume command be used.

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

*No matching move style parameters*  
Style of requested move does not exist.

*No particle species match dump species*  
Requested dump species does not exist.

*Part,spec %d,%d on step %d exceeds MAXITER*  
Movement of particle has exceeded preset iteration count. This error should not occur unless there is a problem with the move.

*Particle %d is outside global domain*  
Particle being read from file is outside the global domain.

*Particle %d's triangle does not exist*  
Could not successfully assign a read-in particle to a triangle on a 2d surface. Probably because it is too far from any triangle.

*Particle is outside global domain*  
Particle being read from restart file is outside the global domain.

*Particles surf-ID does not exist*  
2d particles read from file are assigned to a surface ID that does not exist.

*Permeability is set for 2d species*  
Permeability can only be set for 3d species.

*Permeable probabilities do not sum to 1.0*  
Self-explanatory.

*Permeable stick species is not 2d*  
When a 3d species sticks to a surface, it must become a 2d species.

*Post-migrate: %d particles on proc %d are not in correct bin*  
Particle migration to a new processor has resulted in incorrect bin assignments. This error should not occur.

*Power by 0 in variable formula*  
Self-explanatory.

*Pre-migrate: %d particles on proc %d are not in correct bin*  
Particle migration to a new processor is occurring with incorrect bin assignments. This error should not occur.

*Probability %g for reaction %s is too large*  
The computed probability exceeds 1.0. You need to reduce the reaction distance so the probability will shrink.

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

*Reaction ID %s already exists*  
A reaction ID can only be used once.

*Reaction cannot have more than MAX\_PRODUCT products*  
This is a hard limit in react.cpp.

*Reaction has no numeric rate*

Self-explanatory.

*Reaction must have 0,1,2 reactants*

Self-explanatory.

*Reading 2d particles for a 3d species*

3d particles cannot be assigned to a surface.

*Reading 3d particles for a 2d species*

2d particles must be assigned to a surface.

*Region extends outside global domain*

Self-explanatory.

*Region intersects bin next to periodic boundary*

For a periodic dimension, there must be a bin at each boundary which has no region intersecting it.

*Region surf-ID already exists*

Surface IDs must be unique.

*Replacing a fix, but new style != old style*

A fix ID can be used a 2nd time, but only if the style matches the previous fix. In this case it is assumed you wish to reset a fix's parameters. This error may mean you are mistakenly re-using a fix ID when you do not intend to.

*Reuse of dump ID*

A dump ID cannot be used twice.

*Run style does not support checking*

Not all run styles can use check command.

*Simulation domain is already set*

Cannot use global command twice.

*Species already exists and no new aliases are defined*

Cannot use a species name more than once.

*Sqrt of negative in variable formula*

Self-explanatory.

*Sticking species is not 2d*

A membrane-bound species must be defined as dimension = 2d.

*Substitution for illegal variable*

Input script line contained a variable that could not be substituted for.

*Summed probability %g for species %s & %s is too large*

Probabilities for all reactions involving this pair of species > 1.0

*Summed probability %g for species %s is too large*

Probabilities for all reactions involving this species > 1.0

*Triangle intersects bin next to periodic boundary*

For a periodic dimension, there must be a bin at each boundary which has no triangles intersecting it.

*Two or more wildcard \* in count species ID*

A string with a wildcard character "\*" can only have one wildcard.

*Two or more wildcard \* in diffusion species ID*

A string with a wildcard character "\*" can only have one wildcard.

*Two or more wildcard \* in dimension species ID*

A string with a wildcard character "\*" can only have one wildcard.

*Two or more wildcard \* in stats species ID*

A string with a wildcard character "\*" can only have one wildcard.

*Unbalanced quotes in input line*

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

*Unexpected end of file*

ChemCell hit the end of the file while attempting to read particles or surface info. Something is wrong with the format of the file.

*Universe/uloop variable count < # of partitions*

A universe or uloop style variable must specify a number of values  $\geq$  to the number of processor partitions.

*Unknown command: %s*

The command is not known to ChemCell. Check the input script.

*Unknown species %s in check command*

Self-explanatory.

*Unknown species %s in permeable command*

Self-explanatory.

*Unknown species in count command*

Self-explanatory.

*Unknown species in diffusion command*

Self-explanatory.

*Unknown species in dimension command*

Self-explanatory.

*Unknown species in move\_test command*

Self-explanatory.

*Unknown species in particles command*

Self-explanatory.

*Unknown species in reaction command*

Self-explanatory.

*Unknown surface %s in check command*

Self-explanatory.

*Unknown surface %s in permeable command*

Self-explanatory.

*Variable name must be alphanumeric or underscore characters*

Self-explanatory.

*Vertex %d in surf %s is outside global domain*

One or more vertices in triangles being read in are outside global domain.

*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:**

*Particle is inside surface*

Check determined a particle is inside surface when it should be outside.

*Particle is outside surface*

Check determined a particle is outside surface when it should be inside.

*Restart file used different # of processors*

The restart file was written by a simulation run on a different number of processors.

*Restart file version does not match ChemCell version*

The restart file was written with an older version of ChemCell.

*Run styles do not match*

The restart file was written by a simulation that defined a different run style.

## balance command

### Syntax:

balance style arg(s) file

- style = *static* or *dynamic*

```
static args: bin or particle
    bin = balance by bin count
    particle = balance by particle count
dynamic args: N threshold
    N = test whether to balance every N steps
    threshold = only balance if imbalance is greater than threshold
file = optional filename for storing sub-domain boundaries
```

### Examples:

```
bin static particle
bin dynamic 1000 1.5 out.file
```

### Description:

For parallel runs of spatial simulations, determine how bins are partitioned across processors.

If this command is not used, the set of 3d bins are partitioned into small 3d bricks, one per processor.

If "static bin" is used, bins are partitioned by recursive coordinate bisectioning (RCB) one time at the beginning of the 1st run. Each bin is weighted the same, no matter how many particles it contains.

If "static particle" is used, bins are partitioned by recursive coordinate bisectioning (RCB) at the beginning of each run. Each bin is weighted by the number of particles it contains.

If "dynamic" is used, bins are re-partitioned every N timesteps, with each bin weighting by their current number of particles. Re-partitioning only occurs if the imbalance factor exceeds the specified threshold. The imbalance factor is the ratio of the maximum particles on any processor divided by the average particles across all processors, so an imbalance factor of 1.0 is perfect balance.

File is an optional argument. If specified the load balance partitions are written to the file each time the balancer is invoked, formatted as a surface of triangles which can be visualized with the [Pizza.py toolkit](#).

### Restrictions:

This command can only be used for spatial simulations.

**Related commands:** none

### Default:

Static partitioning of bins as 3d bricks.

## bin command

### Syntax:

bin keyword value(s) ...

- one or more keyword/value pairs may be appended
- valid keywords: *diff* or *react* or *size* or *count*

```
diff value = D
    D = diffusion coeff (cm^2/sec)
react value = dist
    dist = distance (microns)
size values = Lx Ly Lz
    Lx,Ly,Lz = bin sizes in each dimension (microns)
count values = Nx Ny Nz
    Nx,Ny,Nz = bin count in each dimension
```

### Examples:

bin diff 1.0e-8 bin react 0.02 diff 1.0e-7 bin count 30 35 40

### Description:

Create a set of bins that overlay the simulation domain. Bins are used for storing the location of particles and geometry elements (regions, surface triangles) so they can be efficiently located.

Bin size is determined by one or more keyword/value pairs. The actual bin size is set to the maximum of the values induced by each keyword.

The *diff* keyword sets the bin size to the maximum distance a particle will move in one timestep using the specified D and the [timestep](#) and [move\\_style](#) settings. You may need to run ChemCell once to see the move distances printed out at setup.

The *react* keyword sets the bin size by the specified distance, assumed to be the largest distance at which a binary reaction between 2 particles will occur. You may need to run ChemCell once to see the reaction distances printed out at setup.

The *size* keyword sets the bin size in each dimension to the specified values.

The *count* keyword sets the # of bins in each dimension to the specified values.

Since the global domain is sub-divided by an integer # of bins in each dimension, the actual bin size chosen may be slightly larger than the values determined by the *diff*, *react*, or *size* keywords.

### Restrictions:

This command can only be used for spatial simulations.

The [global](#) command must be used before this command, to specify the extent of the spatial simulation domain.

**Related commands:** none



**Default:** none

## boundary command

### Syntax:

```
boundary x y z
```

- $x, y, z = p$  or  $n$

$p$  is periodic

$n$  is non-periodic

### Examples:

```
boundary p p n
```

### Description:

Set the periodicity of the global domain in each of the x,y,z dimensions. "N" means the domain is non-periodic in that dimension, so that if particles move outside the domain, they are lost. "P" means the domain is periodic in that dimension, so particles wrap-around to the other size and can react with nearby particles across the periodic boundary.

**Restrictions:** none

**Related commands:**

[global](#)

### Default:

All dimensions are periodic.

## check command

### Syntax:

```
check species-ID surf-ID flag lo hi plo phi
```

- species-ID = particle species to check
- surf-ID = surface to check
- flag = *in* or *out* or *none*
- lo,hi = optional timestep bounds
- plo,phi = optional particle index bounds

### Examples:

```
check Ca_cyto ER_surf out 1000 2000
```

### Description:

Check particles of species-ID to see if they are inside or outside of the surface with surf-ID. A warning will be printed if a particle does not satisfy the check condition. This can be useful for debugging purposes to test if particles have leaked thru a boundary they shouldn't have and to identify on what timestep the leakage occurred.

The flag determines whether the check is for the particles being inside or outside the surface. A value of *none* means do not perform the check. This command can be used multiple times to specify combinations of particle species and surfaces to check.

The surf-ID is assumed to be a closed set of triangles or closed region. Performing a check for a triangulated surface is expensive; all triangles are looped over for each particle.

Lo/hi are optional timestep values between which (inclusive) the check is done. If not specified, a check is performed every timestep.

Plo/phi are optional particle indices (0 to N-1) to specify a subset of particles (inclusive) to perform the check on. If not specified all particles (of the specified species) are checked. Note that lo/hi must be specified in order to also specify plo/phi.

### Restrictions:

Only 3d particles can be checked.

### Related commands:

[debug](#)

### Default:

No particle-surface checks are performed.

## clear command

### Syntax:

```
clear
```

### Examples:

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

### Description:

This command deletes all particles, restores all settings to their default values, and frees all memory allocated by ChemCell. Once a clear command has been executed, it is as if ChemCell 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-ID N
```

- species-ID = particle species to populate
- N = number of particles for stochastic models or concentration (molarity) for ODE models

### Examples:

```
count Ca 1000  
count receptor* 500
```

### Description:

Set the initial particle count for a non-spatial models. For stochastic models (Gillespie), N is the particle count. For deterministic models (ODEs), N is the particle concentration in molarity.

The species-ID can contain a single wildcard character \* which will match species and alias names in the usual way. E.g. species-ID can be \*, ab\*, \*ab, ab\*cd.

**Restrictions:** none

This command can only be used for non-spatial simulations.

### Related commands:

[particle](#)

### Default:

Count = 0 for all species.

## debug command

### Syntax:

```
debug proc-ID timestep index
```

- proc-ID = processor (0 to P-1) to perform debug testing on
- timestep = timestep to do testing on
- index = particle index (0 to N-1) to perform testing on

### Examples:

```
debug 0 145 7555
```

### Description:

Track the diffusive motion of a single particle on a single processor during a single timestep. This enables a print out of its interactions with regions and triangles for debugging purposes.

To turn off debugging for a subsequent run, set proc-ID to -1.

### Restrictions:

In order to use this command, ChemCell must be compiled with this line uncommented at the top of src/move.cpp:

```
#define DEBUG_MOVE
```

or with a -DDEBUG\_MOVE switch defined in your Makefile.foo.

### Related commands:

[check](#)

**Default:** none

## diffusion command

### Syntax:

```
diffusion species-ID value
```

- species-ID = particle species
- value = diffusion coefficient (cm<sup>2</sup>/sec)

### Examples:

```
diffusion Ca 0.0  
diffusion A* 1.0e-6
```

### Description:

Set the diffusion coefficient for one or more particle species. A value of 0.0 means the particles will be stationary.

The species-ID can contain a single wildcard character \* which will match species and alias names in the usual way. E.g. species-ID can be \*, ab\*, \*ab, ab\*cd.

### Restrictions:

This command can only be used for spatial simulations.

**Related commands:** none

### Default:

Diffusion coefficient = 0.0 for all species.

## dimension command

### Syntax:

```
dimension species-ID value
```

- species-ID = particle species
- value = dimensionality of species = 2 or 3

### Examples:

```
dimension Ca 3  
dimension receptor* 2
```

### Description:

Set the dimensionality for one or more particle species. A value of 3 means the particles diffuse volumetrically. A value of 2 means the particles diffuse on a surface; see the [region](#) or [triangles](#) command.

The species-ID can contain a single wildcard character \* which will match species and alias names in the usual way. E.g. species-ID can be \*, ab\*, \*ab, ab\*cd.

### Restrictions:

The dimensionality of a species cannot be changed if particles of that species already exist.

This command can only be used for spatial simulations.

**Related commands:** none

### Default:

Dimension = 3 for all species.



## dump command

### Syntax:

```
dump ID N filename species1-ID species2-ID ...  
dump ID delta filename species1-ID species2-ID ...
```

- ID = user–assigned name for the dump
- N = dump particles every N timesteps (integer value)
- delta = dump particles every delta seconds (floating point value)
- filename = file to dump to
- speciesN–ID = optional list of species–IDs to dump to file

### Examples:

```
dump 1 100 tmp.dump  
dump Ca-dump 0.1 tmp.dump.Ca Ca-cyto Ca-ER
```

### Description:

Dump a snapshot of particle coordinates and species types to a file every so often as a simulation runs. The species list is optional; if not specified, all particles are dumped.

The ID of a dump can contain alphanumeric characters and underscores.

The frequency of output can be specified in timesteps or elapsed simulation time. Using an integer versus a floating–point value (with a decimal point) determines how the argument is interpreted.

Any species–ID can contain a single wildcard character \* which will match species and alias names in the usual way. E.g. a species–ID can be \*, ab\*, \*ab, ab\*cd.

Multiple dumps (with different IDs) can be defined. See the [undump](#) command for turning off a dump.

The format of the dump file is self–explanatory. For each particle, an ID, its species type (an integer), and its x,y,z coordinates are printed. Dump files can be visualized with the [Pizza.py toolkit](#) or converted into other formats (via Pizza.py) suitable for visualization with other tools, e.g. [VMD](#).

### Restrictions:

This command can only be used for spatial simulations.

### Related commands:

[dump\\_modify](#), [stats](#), [undump](#)

**Default:** none

## dump\_modify command

### Syntax:

```
dump_modify dump-ID keyword args ...
```

- dump-ID = ID of dump to modify
- one or more keyword/arg pairs may be appended
- keyword = *orient*

*orient* arg = *yes* or *no*

### Examples:

```
dump_modify 1 orient yes
```

### Description:

Modify the parameters of a previously defined dump command.

The *orient* keyword determines whether 3 orientation values are added to each particle in the dump file. These could be used for visualization purposes. For 2d species, the 3 values will be the normal vector components for the current location of the particle on its 2d surface (region or triangulated surface). For 3d species, the 3 values will be 0.0.

**Restrictions:** none

**Related commands:**

[dump](#)

### Default:

The option defaults are orient = no.

## echo command

### Syntax:

```
echo style
```

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

### Examples:

```
echo both
echo log
```

### Description:

This command determines whether ChemCell echoes each input script command to the screen and/or log file as it is read and processed. If an input script has errors, it can be useful to look at echoed output to see the last command processed.

The [command-line switch](#) `-echo` can be used in place of this command.

**Restrictions:** none

**Related commands:** none

### Default:

```
echo log
```

## fix command

### Syntax:

```
fix ID style args
```

- ID = user–assigned name for the fix
- style = style name of fix (see possible styles below)
- args = arguments used by a particular style

### Examples:

```
fix 1 conc 100 data.file 2 CheA CheAa 2 CheAa CheA
```

### Description:

Set a fix that will be applied during a simulation. In ChemCell, a "fix" is any operation that is applied to the system during timestepping. Examples include adjusting reaction rates, applying boundary conditions, or computing diagnostics. There is currently only a small number of fixes defined in ChemCell, but others can be added – see [this section](#) of the documentation for a discussion.

The ID of a fix can contain alphanumeric characters and underscores.

Each fix style has its own documentation page which describes its arguments and what it does. For example, see the [fix conc](#) page for information on style *conc*.

Fixes perform their operations at different stages of the timestep. If 2 or more fixes both operate at the same stage of the timestep, they are invoked in the order they were specified in the input script.

Specifying a new fix with the same ID as an existing fix effectively replaces the old fix (and its parameters) with the new fix. This can only be done if the new fix has the same style as the existing fix.

Fixes can be deleted with the [unfix](#) command. Note that this is the only way to turn off a fix; simply specifying a new fix with a similar style will not turn off the first one.

Here is an alphabetic list of fix styles currently defined in ChemCell:

- [fix conc/random](#) – set the concentration of one or more species
- [fix conc/set](#) – set the concentration of one or more species
- [fix dna/toggle](#) – model DNA toggling on/off to modulate mRNA production
- [fix rate/saturate](#) – adjust the rate of one or more reactions

**Restrictions:** none

**Related commands:**

[unfix](#)

**Default:** none

## fix conc/random command

### Syntax:

```
fix ID conc/random N file Nin insp1 insp2 ... Nout outsp1 outsp2 ...
```

- ID is documented in [fix](#) command
- conc/random = style name of this fix command
- N = apply fix every this many timesteps
- file = filename in which time-course data is specified
- Nin = # of input species
- insp1,insp2,etc = IDs of input species
- Nout = # of output species
- outsp1,outsp2,etc = IDs of output species

### Examples:

```
fix mine conc/random 100 data.time 2 CheA CheAa 2 CheAa CheA
```

### Description:

Randomize the species type of certain particles at specified time increments in a spatial simulation. This is an effective way of coupling ChemCell to another model or simulation that pre-computed concentration vs time profiles of certain species.

Here is how the fix operates. Every N timesteps, the list of current particles is scanned. Each particle with a species type in the list of Nin input species has its species type reset to a new value which is one of the Nout output species. Each of the Nout species has a fractional probability associated with it (between 0.0 and 1.0), which sum to 1.0. The assignment of a particle to a new species is done randomly, in accord with those probabilities.

The probabilities for new species can be time-dependent and are read-in from the time-course data file, which has the following format. Lines beginning with a "#" character are ignored. Other lines must begin with a time value (in seconds), followed by Nout-1 values (between 0.0 and 1.0) which sum to a value  $\leq 1.0$ . These are the probabilities for each of the Nout species. The value for the last Nout species is set to  $1.0 - \text{sum}$ . The time stamp for successive lines in the file should be monotonically increasing.

On a timestep when the fix is applied, the file line with a time stamp just smaller (or equal) to the current time is used to set the Nout probabilities.

### Restrictions:

This fix can only be used with spatial simulations.

### Related commands:

[fix conc/set](#)

**Default:** none

## fix conc/set command

### Syntax:

```
fix ID conc/set N file Nsp sp1 sp2 ...
```

- ID is documented in [fix](#) command
- conc/random = style name of this fix command
- N = apply fix every this many timesteps
- file = filename in which time–course data is specified
- Nsp = # of species to set concentration of
- sp1,sp2,etc = IDs of species

### Examples:

```
fix mine conc/set 100 data.time 2 CheA CheAa
```

### Description:

Set the concentration level of certain species at specified time increments in a non–spatial simulation. This is an effective way of coupling ChemCell to another model or simulation that pre–computed concentration vs time profiles of certain species.

Here is how the fix operates. Every N timesteps, the concentration or count of particles of each species in the specified list is reset to a new value which is listed in the time–course data file, which has the following format. Lines beginning with a "#" character are ignored. Other lines must begin with a time value (in seconds), followed by Nsp values. The time stamp for successive lines in the file should be monotonically increasing. For stochastic simulations (Gillespie), the value for each species is a particle count. For continuum simulations (ODE), the value for each species is a concentration in molarity.

On a timestep when the fix is applied, the file line with a time stamp just smaller (or equal) to the current time is used to set the species concentrations.

### Restrictions:

This fix can only be used with non–spatial simulations.

### Related commands:

[count](#), [fix conc/random](#)

**Default:** none

## fix dna/toggle command

### Syntax:

```
fix ID dna/toggle N speciesDNA-ID Kon Koff reactRNA-ID Ktranscription Kconstitutive reactDNA-ID bind-ID
```

- ID is documented in [fix](#) command
- dna/toggle = style name of this fix command
- N = apply fix every this many timesteps
- speciesDNA-ID = ID of DNA species
- Kon = rate at which DNA is turned on (per second per molarity)
- Koff = rate at which DNA is turned off (per second)
- reactRNA-ID = ID of reaction which produces mRNA transcripts
- Ktranscription = transcription rate at which mRNA is produced (per second per molarity)
- Kconstitutive = constitutive rate at which mRNA is produced (molarity per second)
- reactDNA-ID = ID of reaction which represents DNA on/off state
- bind-ID = ID of transcription species which binds to DNA
- Vratio = volume ratio to use as scale factor on bind-ID concentration

### Examples:

```
fix mine dna/toggle 1 DNA 2.0e5 0.02 RNA-reac 100.0 0.1 DNA-reac NFkB-nuc 10.0
```

### Description:

Adjust two reaction rates based on a DNA species which toggles on and off. This is a method for modeling the state of a DNA site as occupied (on) by a transcription factor or unoccupied (off). When occupied, mRNA transcripts are produced at a transcription rate; when unoccupied they are produced at a 2nd constitutive rate.

Here is how the fix operates. Assume a speciesDNA-ID is defined. Also assume 2 reactions are defined. The first is reactRNA-ID which produces mRNA transcripts. The second is reactDNA-ID which toggles the DNA on and off. They should be specified in the following form where NULL means there are no reactants. As discussed below, their specified rates are ignored since the rates are set by this fix, so they can be specified as 0.0.

```
NULL -> mRNA
NULL -> speciesDNA-ID
```

```
reaction reactRNA-ID 0.0 mRNA-ID
reaction reactDNA-ID 0.0 speciesDNA-ID
```

Every N timesteps, two operations are performed. First the concentration or count of speciesDNA-ID is reset. For continuum ODE simulations it will be a continuous value between 0 and 1. For stochastic models, it will be a discrete count, either 0 or 1. A value of 0 represents an "off" or unbound state for the DNA, while 1 represents an "on" or bound state. As discussed below, in stochastic models, the 2nd reaction will potentially set the DNA count to 1 or 2; the latter value will be changed to 0 by the fix. I.e. the DNA toggles on or off.

Second, the rates of the 2 reactions are reset in the following way. The rate of the 1st reaction is set to be

$$K_{\text{new}} = K_{\text{transcription}} * K_{\text{off}}/K_{\text{on}} * \text{DNA} + K_{\text{constitutive}} * (1 - \text{DNA})$$

Ktranscription and Kconstitutive are the parameters specified in the fix command, as are Koff and Kon.

$K_{\text{transcription}}$  is the rate at which mRNA is produced when the transcription factor is bound to the DNA site.  $K_{\text{constitutive}}$  is the native rate mRNA is produced when the DNA site is unbound.  $K_{\text{off}}/K_{\text{on}}$  has units of molarity and represents an average concentration.  $\text{DNA}$  is a unitless number between 0 and 1. The units of both terms and thus the overall rate  $K_{\text{new}}$  is molarity/sec which is the rate at which the mRNA species in the 1st reaction will be produced.

The rate of the 2nd reaction is set to be

$$K_{\text{new}} = \text{sign} * K_{\text{off}} * \text{DNA} + K_{\text{on}} * [\text{bind-ID}] * (1 - \text{DNA})$$

$K_{\text{off}}$ ,  $K_{\text{on}}$ , and  $\text{DNA}$  are as above.  $[\text{bind-ID}]$  is the concentration of the transcription factor species which binds to the DNA site. This concentration will be boosted by  $V_{\text{ratio}}$  if you need to account for the fact that the transcription factor species is in a smaller-volume compartment, e.g. the nucleus. This is only relevant for stochastic models since the concentration should already be scaled appropriately in an ODE model. The units of  $K_{\text{new}}$  are thus per second which is the rate at which the DNA toggles its state.

$\text{Sign}$  is set to a value of  $-1$  for continuum ODE models. This is what keeps the value of  $\text{DNA}$  between 0 and 1.  $\text{Sign}$  is set to  $+1$  for stochastic models. If this reaction is performed when  $\text{DNA}$  is on (value of 1), then the  $\text{DNA}$  count will increase from 1 to 2, and be reset to 0 on the subsequent timestep as described above. Thus the  $\text{DNA}$  will effectively be turned off.

**Restrictions:**

This fix can only be used with non-spatial simulations.

**Related commands:** none

**Default:** none



## fix rate/saturate command

### Syntax:

```
fix ID rate/saturate N species-ID half volscale react1-ID react2-ID ...
```

- ID is documented in [fix](#) command
- rate/saturate = style name of this fix command
- N = apply fix every this many timesteps
- species-ID = ID of species whose dynamic concentration will affect rate
- half = concentration of species-ID which will cut rate in half (molarity)
- volscale = scale factor to apply to concentration of species-ID
- react1-ID, react2-ID, ... = list of reactions whose rates are affected

### Examples:

```
fix mine rate/saturate 1 NFkB-nuc 100 c1 u1 u3
```

### Description:

Adjust one or more reaction rates based on the current concentration of a chosen species. This is a way to have a time-dependent rate for a reaction that goes to zero as the concentration of one of its reactants increases, so that the amount of product produced effectively saturates rather than increases forever. For example, it can be used to model gene transcription resulting in mRNA production.

Here is how the fix operates. Every N timesteps, the rates for the listed reactions are scaled to new values in the following way:

$$r_{\text{new}} = r_{\text{initial}} * \text{half} / (\text{half} + [\text{species-ID}])$$

$R_{\text{new}}(t)$  is the new scaled reaction rate.  $R_{\text{initial}}$  is the rate set for this reaction initially in the input script. Half is the concentration value set by this command. [Species-ID] is the current concentration of the selected species.

For continuum ODE simulations this concentration is stored directly by ChemCell and is correct for whatever compartment the species may be in. Thus the *volscale* parameter is ignored. For stochastic models, the concentration is computed by the molecule count divided by the volume. It is then scaled by the specified *volscale* parameter. This should be set to 1.0 if the system volume is the one in which *species-ID* is present. If the concentration is for a species in a smaller volume (e.g. the nucleus), then *volscale* should be set to the ratio of the system volume to the smaller volume, e.g. 10.0 if the cell is 10x larger than the nucleus.

Note that the *half* parameter is the concentration of species-ID at which the effective rate will be cut in half. Note that the scale factor  $\text{half} / (\text{half} + [\text{species-ID}])$  is unitless.

### Restrictions:

This fix can only be used with non-spatial simulations.

**Related commands:** none

**Default:** none

## global command

### Syntax:

```
global xlo ylo zlo xhi yhi zhi
```

- xlo,ylo,zlo = lower-left corner of simulation domain (microns)
- xhi,yhi,zhi = upper-right corner of simulation domain (microns)

### Examples:

```
global 0 0 0 5 10 5
```

### Description:

Define the lower-left and upper-right corners of the global simulation domain for a spatial simulation. All particles and cellular geometry must be inside this domain.

### Restrictions:

This command can only be used for spatial simulations. See the [volume](#) command for non-spatial simulations.

### Related commands:

[volume](#)

**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 ChemCell 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 ChemCell could run for a long time.

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

**Restrictions:** none

### Related commands:

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

**Default:** none

## jump command

### Syntax:

```
jump file label
```

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

### Examples:

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

### Description:

This command closes the current input script file, opens the file with the specified name, and begins reading ChemCell 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
dump 1 all atom 100 file.$a
run 10000
undump 1
next a
jump in.lj loop
```

If the jump *file* argument is a variable, the jump command can be used to cause different processor partitions to run different input scripts. In this example, ChemCell 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 ccell_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, ChemCell will come to the end of the file and exit.

### Related commands:

[variable](#), [include](#), [label](#), [next](#)

**Default:** none

## label command

### Syntax:

```
label ID
```

- ID = string used as label name

### Examples:

```
label xyz  
label loop
```

### Description:

Label this line of the input script with the chosen ID. Unless a jump command was used previously, this does nothing. But if a [jump](#) command was used with a label argument to begin invoking this script file, then all command lines in the script prior to this line will be ignored. I.e. execution of the script will begin at this line. This is useful for looping over a section of the input script as discussed in the [jump](#) command.

**Restrictions:** none

**Related commands:** none

**Default:** none

## log command

### Syntax:

```
log file
```

- file = name of new logfile

### Examples:

```
log log.equil
```

### Description:

This command closes the current ChemCell 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.ccell" is the default log file for a ChemCell 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 ChemCell log file is named log.ccell.

## move\_style command

### Syntax:

```
move_style option1 option2 ...
```

- options = *cube* or *sphere* or *square* or *circle* or *uniform* or *brownian*

### Examples:

```
move_style sphere brownian  
move_style circle uniform
```

### Description:

Set one or more options that determine how particles move in a spatial simulation.

The *cube* option means particle will sample points in a small cube around their locations. The *sphere* option means particles will sample points in a small sphere around their location.

The *square* option means particles will only move in 2d and sample points in a small xy square around their locations. The *circle* option means particles will only move in 2d and sample points in a small xy circle around their locations. These options can be used with a [global](#) simulation domain that is small in the z-dimension to effectively setup a 2d simulation.

The *uniform* option means the sampling of volumetric (or planar) space is uniform. The *brownian* option means the sampling is from a Gaussian so that particles move in Brownian motion.

For 2d particles diffusing on regions or triangulated surfaces, only the uniform/brownian options are relevant since particles always effectively move in 2d, sampling within a circle tangential to the surface.

### Restrictions:

This command can only be used for spatial simulations.

**Related commands:** none

### Default:

The default move style is cube brownian.



## move\_test command

### Syntax:

```
move_test species-ID N Nhisto seed file
```

- species-ID = ID of species
- N = # of moves
- Nhisto = number of histogram bins
- seed = random # seed
- file = file to write results to

### Examples:

```
move_test Ca 10000 100 58327 tmp.diff
```

### Description:

Test the diffusive movement of a species by performing N moves of a single particle, and binning the result in distance. The histogram statistics are written to the specified file.

This is useful for testing the distribution of move distances for a particular diffusion coefficient and timestep size.

**Restrictions:** none

**Related commands:**

[move\\_style](#)

**Default:** none

## 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 ChemCell 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. *Atom*- or *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 ChemCell 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.ccell.variable" and "tmp.ccell.variable.lock" which you will see in your directory during such a ChemCell 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.abc, 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
include abc.model
run 10000
shell cd ..
clear
next d
jump in.abc
```

If the variable "d" were of style *universe*, and the same in.abc 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
...
include abc.model.$i$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

## particles command

### Syntax:

```
particles species-ID N surface-ID
```

- species-ID = user-assigned species name for particles
- N = number of particles that follow
- surface-ID = optional surface ID to place the particles on

### Examples:

```
particles Ca 10000  
particles receptor 500 membrane
```

### Description:

Read N particles of type species-ID from successive lines of the input script. The species-ID must already have been defined via the [species](#) command. If the species is 3D, do not use the surface-ID argument. If the species is 2d, the surface-ID argument must be used to specify what region or triangulated surface the particles are on. Note that the species dimensionality must have already been set (default or via the [dimension](#) command) before particles are read in.

Following the particles command, the next line in the input script is skipped (leave it blank). The following N lines should have an index (1–N) and 3 particle coordinates (x,y,z in microns) on each line.

Note that a long list of particles can be put in a separate file and read in via the [include](#) command.

### Restrictions:

The [bin command](#) must be used before defining particles.

This command can only be used for spatial simulations.

**Related commands:** none

**Default:** none

## permeable command

### Syntax:

```
permeable species-ID surface-ID flag keyword value ...
```

- species-ID = ID of particle species
- surface-ID = ID of region or triangulated surface
- flag = *in* or *out* or *both*
- one or more keyword/value pairs can be specified

```
keywords = reflect or near or stick or far or thru or rsp or nsp or ssp or fsp or tsp
reflect, near, stick, far, thru value = probability
probability = value between 0.0 and 1.0 (inclusive)
rsp, nsp, ssp, fsp, tsp value = species-ID
species-ID = what the permeating particle will become if event takes place
```

### Examples:

```
permeable A nucleus out reflect 0.9 thru 0.1 tsp A_nuc
permeable Ca_cyto cell reflect 1.0
```

### Description:

Set the permeability for a species when it encounters a surface (region or triangulated surface) during a move. The *flag* value in/out/both refers to which side of the surface. Each triangle in a triangulated surface has an "outside" determined by applying the right-hand rule to its 3 ordered vertices.

Each time a 3d particle hits a surface, there are 5 possibilities. It can reflect off and continue its move, end its move by being placed EPSILON away from the surface on the near side, stick to the surface and become a 2d species, end its move by being placed EPSILON away from the surface on the far side, or pass thru the surface unimpeded.

Each of the 5 cases is assigned a fractional probability P, the sum of which must be 1.0. When a particle moves and a surface is encountered, a random number is generated which is used to select which of the 5 cases occurs.

Each of the 5 surface interactions can also cause the particle to become a new species, if the corresponding keyword (rsp,nsp,ssp,fsp,tsp) is specified.

### Restrictions:

This command cannot be used for 2d species. If the stick probability is non-zero, the *ssp* keyword must be specified, else the 3d species cannot become a 2d species.

This command can only be used for spatial simulations.

**Related commands:** none

### Default:

By default, all 3d species interact with all surfaces with a reflection probability of 1.0 and with no species change.

## print command

### Syntax:

```
print string
```

- string = text string to print, which may contain variables

### Examples:

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

### Description:

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

If you want the print command to be executed multiple times (with changing variable values), there are 3 options. First, consider using the [fix print](#) command, which will print a string periodically during a simulation. Second, the print command can be used as an argument to the *every* option of the [run](#) command. Third, the print command could appear in a section of the input script that is looped over (see the [jump](#) and [next](#) commands).

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

**Restrictions:** none

**Related commands:**

[fix print](#), [variable](#)

**Default:** none

## probability command

### Syntax:

probability style value

- style = *max* or *diff*

*max* value = Pmax = fraction between 0.0 and 1.0

*diff* value = M = multiplier on sum of diffusive distances

### Examples:

```
probability max 0.5
probability diff 1.5
```

### Description:

Set parameters that determine how binary reactions are computed for spatial simulations. A binary reaction occurs in a timestep with a probability P when a pair of reactant particles are within a cutoff distance R.

The *max* style uses the specified Pmax to set one distance R for all reactions so that the fastest reaction (maximum rate constant) will occur with probability Pmax and give the desired k for a well-mixed system with diffusion-limited reactions. Since other reactions happen less frequently (smaller rate constants) their P values are set smaller than the specified Pmax.

The *diff* style sets a distance R for each reaction based on the diffusion coefficients of its 2 reactants. The sum of the root-mean squared value for each reactant is multiplied by the specified M factor to compute R. The P for each reaction is then set so that each reaction happens at the desired frequency for a well-mixed system with diffusion-limited reactions.

Note that values of P and R for individual reactions can be overridden via the [react\\_modify](#) command.

### Restrictions:

This command can only be used for spatial simulations.

**Related commands:** none

### Default:

max 0.5

## react\_modify command

### Syntax:

```
react_modify reaction-ID keyword value(s) ...
```

- reaction-ID = ID of reaction to modify
- one or more keyword/value pairs can be specified

```
keywords = rate or loc or dist or prob or weight
rate value = rate
    rate = reaction rate (same units as in reaction command)
loc values = product styleflag whichflag dirflag
    product = which product (1-N)
    styleflag = def or at or near
    whichflag = def or 1 or 2 or 1/2
    dirflag = def or in or out or in/out
dist = R
    R = distance at which reactants will react (microns)
prob = P
    P = probability (0-1) with which reactants react
weight values = which index ratio
    which = reactant or product
    index = which reactant or product (1-N)
    ratio = weighting factor to apply
```

### Examples:

```
react_modify 1 dist 0.05 prob 1.0
react_modify channel 2 near 2 out
```

### Description:

Reset one or more parameters of a specific reaction.

The *rate* keyword resets the reaction rate.

The *loc* keyword sets the location at which a product of the reaction is placed. The *product* setting determines which product (1 to N). A value of -1 means all products. The *styleflag* setting refers to placing the product at or near a reactant's location. The *whichflag* setting refers to which reactant (1 to N) to place the product at or near. A value of 1/2 means to randomly choose which reactant each time a reaction occurs. The *dirflag* setting only applies to 3d products placed near 2d reactants and refers to which side of the surface (inside or outside) to place the product on. A value of *in/out* means to randomly choose the side each time a reaction occurs. A value of *def* for default should be used if *dirflag* does not apply.

Depending on the kind of reaction (1 or 2 reactants) and whether the reactants are 3d diffusing species or 2d (on a surface), certain combinations of settings are invalid and will generate errors. Note that only 3d products can be placed at a 3d reactant's location. Only 2d products can be placed at a 2d reactant's location. And only 3d products can be placed near a 2d reactant's location, which means to place it a distance epsilon from the surface the 2d reactant is on.

These are the default rules for reactions with a single reactant:



- If the reactant and product are 3d, put the product at the location of the reactant.
- If the reactant and product are 2d, put the product at the location of the reactant.
- If the reactant is 3d and the product is 2d, this is an error.
- If the reactant is 2d and the product is 3d, put the product near the reactant on the inside of the surface.

These are the default rules for reactions with two reactants:

- If both reactants and the product are 2d, put the product at the location of the reactant with the smaller diffusion coeff.
- If both reactants and the product are 3d, put the product at the location of the reactant with the smaller diffusion coeff.
- If one reactant is 2d and the other 3d and the product is 3d, put the product at the 3d reactant's location.
- If one reactant is 2d and the other 3d and the product is 2d, put the product at the 2d reactant's location.
- If both reactants are 3d and the product is 2d, this is an error.
- If both reactants are 2d and the product is 3d, put the product a distance epsilon away from the reactant with the smaller diffusion coeff on the inside of the surface.
- In all cases where diffusion coeffs are used, if the diffusion coeffs of the 2 reactants are equal, the 1st reactant is used.

The *dist* keyword explicitly sets the cutoff distance for a binary reaction between two reactants. Normally, ChemCell sets this distance itself (see the [probability](#) command for options). Setting the *dist* value to  $-1.0$  turns off the explicit setting; ChemCell will again compute the reaction cutoff.

The *prob* keyword explicitly sets the probability (from 0.0 to 1.0) for a binary reaction to take place (assuming the reactants are within the cutoff distance). Normally, ChemCell sets this probability itself (see the [probability](#) command for options). Setting the *prob* value to  $-1.0$  turns off the explicit setting; ChemCell will again compute the probability.

Note that the *dist* and *prob* keywords give you the option of setting reaction parameters directly, using your own formulation or algorithms and settings from another code. For example, the [Smoldyn](#) simulator calculates a variety of reaction parameterizations for different kinds of reactions, as described in ([Andrews](#)). Essentially it computes reaction cutoff distances (binding radii in Smoldyn nomenclature) as a function of diffusion coefficients of the reactants, timestep, and reaction rate. The Smoldyn formulation spans the spectrum from diffusion-limited to reaction-limited reactions and thus produces reaction cutoff distances which are more accurate and complete than the simple heuristic that ChemCell uses by default.

The *weight* keyword sets a volume weighting factor for a specific reactant or product which adjusts its concentration the reaction occurs and the species is removed or produced. This can be used by non-spatial, continuum simulations (ODEs) to weight the effects of a reaction. The weighting factor can be thought of as a volume ratio between 2 compartments, so a factor can be set to enable a multi-compartment ODE solution where each compartment has its own volume.

**Restrictions:** none

The *loc* and *dist* and *prob* keywords only apply to spatial simulations. The *weight* keyword only applies to non-spatial, continuum (ODE) simulations.

**Related commands:**

[probability](#), [reaction](#)

**Default:** none

---

**(Andrews)** Andrews and Bray, Phys Biol, 1, 137–151 (2004).

## reaction command

### Syntax:

```
reaction ID K product1 product2 ...
reaction ID reactant1 K product1 product2 ...
reaction ID reactant1 reactant2 K product1 product2 ...
```

- ID = user-assigned name for the reaction
- reactant1,reactant2 = species-IDs of reactant(s)
- K = reaction rate (see units below)
- productN = species-IDs of zero or more products

### Examples:

```
reaction 1 Ca_cyto IP3R 1.0e7 IP3R Ca_er
reaction decay receptor-active 0.1 receptor-inactive
reaction influx 1.0e-8 Ca
```

### Description:

Define a reaction that turns reactants into products at a specified reaction rate. The reaction can have zero, one, or two reactants and 0 or more products.

The ID of a reaction can contain alphanumeric characters and underscores and dashes.

The species-IDs of all reactants and products must have been previously defined using the [species](#) command.

The units of the reaction rate K are molarity/sec for reactions with 0 reactants, 1/sec for unary reactions, and 1/molarity-sec for binary reactions.

The [react\\_modify](#) command can be used to specify the spatial location of created products as well as other reaction attributes.

### Restrictions:

Reactions with 0 reactants can only be specified for non-spatial simulations.

The maximum number of product species for any one reaction is hard-wired in src/react.cpp via the MAX\_PRODUCT setting. Currently it is set to 5.

### Related commands:

[react\\_modify](#)

**Default:** none

## read\_restart command

### Syntax:

```
read_restart file
```

- file = name of binary restart file to read in

### Examples:

```
read_restart save.10000
```

### Description:

Read in a previously saved set of particles from a restart file. This allows continuation of a previous spatial simulation.

Only particle information is stored in the restart file. Thus before reading a restart file, you should re-define species and geometry information, and must setup bins via the [bin](#) command.

Because restart files are binary, they may not be portable to other machines.

**Restrictions:** none

The [bin](#) command must be used before reading a restart file.

This command can only be used for spatial simulations.

### Related commands:

[write\\_restart](#), [restart](#)

**Default:** none

## region command

### Syntax:

region ID style args

- ID = user–assigned name for the region
- style = *sphere* or *box* or *plane* or *cylinder*
- args = list of arguments for a particular style

```
sphere args = x y z r
    x,y,z = center of sphere (microns)
    r = radius of sphere (microns)
box args = xlo ylo zlo xhi yhi zhi
    xlo,ylo,zlo = lower left corner of box (microns)
    xhi,yhi,zhi = upper right corner of box (microns)
plane args = x y z nx ny nz
    x,y,z = point on plane (microns)
    nx,ny,nz = vector pointing in normal direction to plane
cylinder args = dim c1 c2 r
    dim = x or y or z
    c1,c2 = coords of axis in other 2 dimensions (yz,xz,xy) (microns)
    r = radius of cylinder (microns)
```

### Examples:

```
region sph sphere 0 0 0 10.0
region cell box 0 0 0 4 2 2
```

### Description:

Define a simple geometric region as a surface. Similar to [triangulated surfaces](#), the region surface can be used as a boundary for 3d particles or as a surface on which 2d particles are located and move.

The ID of a region can contain alphanumeric characters and underscores and dashes.

The *plane* style defines an infinite plane which should be axis–aligned if the simulation box is periodic. The *cylinder* style defines an infinite open–ended axis–aligned cylinder.

### Restrictions:

The [bin command](#) must be used before defining a region.

This command can only be used for spatial simulations.

### Related commands:

[triangles](#)

**Default:** none

## restart command

### Syntax:

```
restart 0
restart N root
restart N file1 file2
```

- N = write a restart file every this many timesteps
- root = filename to which timestep # is appended
- file1,file2 = two full filenames, toggle between them when writing file

### Examples:

```
restart 0
restart 1000 ecoli.restart
restart 10000 ecoli.r.1 ecoli.r.2
```

### Description:

Write out a binary restart file every so many timesteps as a run proceeds. A value of 0 means do not write out restart files. Using one filename as an argument will create a series of filenames with a timestep suffix, e.g. the 2nd example above will create ecoli.restart.1000, ecoli.restart.2000, ecoli.restart.3000, etc. Using two filenames will produce only 2 restart files. ChemCell will toggle between the 2 names as it writes successive restart files.

Only particle information is stored in a restart file.

Restart files can be read by a [read\\_restart](#) command to restart a simulation from a particular state. Because the file is binary (to enable exact restarts), it may not be readable on another machine.

### Restrictions:

This command can only be used for spatial simulations.

### Related commands:

[write\\_restart](#), [read\\_restart](#)

### Default:

```
restart 0
```

## run command

### Syntax:

```
run N  
run delta
```

- N = run for N timesteps (integer value)
- delta = run for delta seconds (floating point value)

### Examples:

```
run 1000  
run 5.0
```

### Description:

Run a simulation for N timesteps or delta seconds. If a previous run was made, this command continues the simulation.

The length of the run can be specified in timesteps or elapsed simulation time. Using an integer versus a floating-point value (with a decimal point) determines how the argument is interpreted.

For stochastic non-spatial (Gillespie) simulations, N refers to a number of reactions, since there is one timestep/reaction.

**Restrictions:** none

**Related commands:**

[run\\_style](#)

**Default:** none

## run\_style command

### Syntax:

```
run_style style
```

- style = *spatial* or *gillespie* or *ode* or *ode/rk*

### Examples:

```
run_style spatial
run_style gillespie
```

### Description:

Set the style of simulation that will be run.

The *spatial* option runs a spatial reaction/diffusion model where particles diffuse volumetrically (3d) or on surfaces (2d) via Brownian motion with a diffusion coefficient defined by the [diffusion](#) command. Particles store coordinates (see the [particles](#) command) and geometric surfaces can be defined (see the [triangles](#) and [region](#) commands) for particles to interact with (see the [permeable](#) command).

In spatial simulations, pairs of particles react each timestep with a probability between 0.0 and 1.0 if they are within a cutoff distance of each other. Reactions and associated rates are set via the [reaction](#) command. The [probability](#) or [react\\_modify](#) commands are used to set the probability and cutoff distance for individual reactions. Setting the probability to 1.0 and the cutoff distance to the binding radius computed by the Smoldyn algorithms described in ([Andrews](#)) enables a [Smoldyn-style](#) spatial simulation to be run.

The *gillespie* option runs the "direct method" version of Gillespie's Stochastic Simulation Algorithm (SSA) as described in ([Gillespie](#)), with computational enhancements outlined in ([Gibson/Bruck](#)). Reactions and associated rates are set via the [reaction](#) command. The volume of the system is set via the [volume](#) command. Initial particle counts are set via the [count](#) command. This is a non-spatial simulation, so particle coordinates are not defined, nor is any geometric information.

The *ode* and *ode/rk* options solve the system of coupled reactions with a continuum ODE time integration scheme. For style *ode* a fixed-size timestep is used. For style *ode/rk* an adaptive timestep Runge-Kutta algorithm is used as described in ([Shampine](#)). Reactions and associated rates are set via the [reaction](#) command. The volume of the system is set via the [volume](#) command. Initial particle counts are set via the [count](#) command. This is a non-spatial simulation, so particle coordinates are not defined, nor is any geometric information.

**Restrictions:** none

**Related commands:** none

**Default:** none

---

(**Andrews**) Andrews and Bray, Phys Biol, 1, 137–151 (2004).



**(Gibson/Bruck)** Gibson and Bruck, J Phys Chem A, 104, 1876–1889 (2000).

**(Gillespie)** Gillespie, J Phys Chem, 81, 2340 (1977).

**(Shampine)** Shampine, Watts, and Davenport, SIAM Review, 18, 376–411 (1976).

## seed command

### Syntax:

seed N

- N = integer random number seed (positive integer)

### Examples:

seed 5982983

### Description:

Set the random # seed used for the simulation. For stochastic non-spatial (Gillespie) simulations, random numbers are used to pick reactions and timesteps. For spatial simulations, random numbers are assigned to particles when they are read in and used thereafter to compute diffusive motion and reaction probabilities.

### Restrictions:

For spatial simulations, this command must be used before particles are read in, else it will have no effect.

**Related commands:** none

### Default:

If not specified, the seed is set by the current system time when the simulation is run. This is a valid random seed, but means the seed will be different the next time the input script is run.

## 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 ChemCell 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:

ChemCell 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

## species command

### Syntax:

```
species species-ID alias1 alias2 ...
```

- species-ID = user-assigned name for the particle
- alias1,alias2,etc = optional alias names by which the species can also be referenced

### Examples:

```
species Ca_cyto  
species C8x432 Ca Ca_input Ca_known
```

### Description:

Define a particle species and alternate names by which it can be referred to in any command that takes a species-ID.

The ID of a species and its aliases can contain alphanumeric characters and underscores and dashes.

If species-ID does not exist, it is created with default dimension, diffusivity, and permeability settings. If species-ID already exists, then additional unique aliases must be specified which are associated with that species-ID.

**Restrictions:** none

**Related commands:** none

**Default:** none

## stats command

### Syntax:

```
stats N species1-ID species2-ID ...  
stats delta species1-ID species2-ID ...
```

- N = print stats every N timesteps (integer value)
- delta = print stats every delta seconds (floating point value)
- speciesN-ID = optional list of species IDs

### Examples:

```
stats 100  
stats 0.01 Ca_cyto Ca_er
```

### Description:

Print particle statistics to the screen (and log file) every so often as a simulation runs. Setting N = 0 will only print statistics at the beginning and end of the run. The units for the particle stats are specified by the [stats\\_modify](#) command. The default is particle count.

The frequency of output can be specified in timesteps or elapsed simulation time. Using an integer versus a floating-point value (with a decimal point) determines how the argument is interpreted.

For stochastic non-spatial (Gillespie) simulations, N refers to a number of reactions, since there is one timestep/reaction.

The list of species is optional. If not specified, counts for all species will be printed.

Any species-ID can contain a single wildcard character \* which will match species and alias names in the usual way. E.g. a species-ID can be \*, ab\*, \*ab, ab\*cd.

**Restrictions:** none

**Related commands:**

[dump](#), [stats\\_modify](#)

**Default:**

N = 0

## stats\_modify command

### Syntax:

```
stats_modify keyword args ...
```

- one or more keyword/arg pairs may be appended
- keyword = *units* or *format*

```
units arg = count or molarity or um or nm
count = integer count of particles
molarity = concentration in molarity = moles/liter
um = concentration in micro-molar
nm = concentration in nano-molar
format arg = C-style format string
```

### Examples:

```
stats_modify units molarity
stats_modify format %7.2g
```

### Description:

Set options for how statistics are printed to the screen and logfile. via the [stats](#) command.

The *units* keyword sets the style of units used to print particle statistics. Style *count* is the only allowed option for spatial simulations, since the enclosing volume is arbitrary. For non-spatial simulations, output can be in concentration units (molarity, uM, nM) since a volume is explicitly specified.

The *format* keyword sets the precision for how each particle species is printed. For unit style *count* this should be an integer setting, e.g. %10d. For other unit styles, it should be a floating point setting, e.g. %7.2g.

**Restrictions:** none

**Related commands:**

[stats](#)

### Default:

The option defaults are units = count, format = %d (for count units) or format = %g (for any of the molarity units).

## timestep command

### Syntax:

```
timestep dt
```

- dt = timestep size (seconds)

### Examples:

```
timestep 1.0e-5  
timestep 0.003
```

### Description:

Set the timestep size for spatial or non-spatial ODE simulations.

### Restrictions:

This command cannot be used for a stochastic non-spatial (Gillespie) simulation, since it chooses the timestep for each reaction itself.

### Related commands:

[run](#), [run\\_style](#)

**Default:** none



## triangles command

### Syntax:

```
triangles surf-ID M N
```

- surf-ID = user-assigned name for triangulated surface
- M = number of vertices that follow
- N = number of triangles and edge connections that follow

### Examples:

```
triangles cell 150 100  
triangles ER 27000 16000
```

### Description:

Define a surface with M vertices and N triangles from successive lines of the input script. If surf-ID already exists, the new triangles are added to it. If it doesn't exist, a new surface is created with default permeability settings for all species.

The ID of a surface can contain alphanumeric characters and underscores and dashes.

Following the triangles command, the next line in the input script is skipped (leave it blank). 3 sections of information follow, separated by blank lines.

The first section has M vertex lines, each with an index (1–M) and 3 coordinates (x,y,z in microns).

The second section has N lines, each with an index (1–N) and 3 integer vertex indices (1–M) which define the corner points of each triangle. The 3 indices should be ordered so that applying the right-hand rule yields a normal vector pointing in the "outward" direction from the triangle face.

The third section has an index (1–N) followed by 6 integers: i ei j ej k ek. I,J,K are the 3 triangles this triangle is connected to across its 3 edges where its 1st edge is between vertices 1–2, its 2nd edge is between 2–3, and its 3rd edge is between 3–1. I,J,K are integer values between 1–N. Ei,Ej,Ek is the edge (1–3) on the other triangle that this edge is connected to. If a triangle has no connection across a particular edge, then the 2 associated values (e.g. I,Ei) should be 0.

Note that a long list of triangles can be put in a separate file and read in via the [include](#) command.

### Restrictions:

The [bin command](#) must be used before defining a triangulated surface.

### Related commands:

[region](#)

**Default:** none

## 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 so that it is no longer active. This closes the file associated with the dump.

**Restrictions:** none

### Related commands:

[dump](#)

**Default:** none

## unfix command

### Syntax:

```
unfix fix-ID
```

- fix-ID = ID of a previously defined fix

### Examples:

```
unfix 2  
unfix influx
```

### Description:

Turn off a fix that was previously defined with a [fix](#) command.

**Restrictions:** none

### Related commands:

[fix](#)

**Default:** none

## unreact command

### Syntax:

```
unreact reaction-ID
```

- reaction-ID = ID of previously defined reaction

### Examples:

```
unreact AB  
unreact channel-flux
```

### Description:

Delete a previously defined reaction so that it is no longer active.

**Restrictions:** none

**Related commands:** none

[react](#)

**Default:** none

## variable command

### Syntax:

variable name style args ...

- name = name of variable to define
- style = *index* or *loop* or *world* or *universe* or *uloop* or *equal* or *atom*

```

index args = one or more strings
loop args = N = integer size of loop
world args = one string for each partition of processors
universe args = one or more strings
uloop args = N = integer size of loop
equal args = one formula containing numbers, math operations, variable references
  numbers = 0.0, 100, -5.4, 2.8e-4, etc
  math operations = (), -x, x+y, x-y, x*y, x/y, x^y,
                  sqrt(x), exp(x), ln(x), log(x),
                  sin(x), cos(x), tan(x), asin(x), acos(x), atan(x),
                  ceil(x), floor(x), round(x)
other variables = v_abc, v_n

```

### Examples:

```

variable x index run1 run2 run3 run4 run5 run6 run7 run8
variable LoopVar loop $n
variable MyValue equal 5.0*exp(v_energy/(v_boltz*v_Temp))
variable beta equal v_temp/3.0
variable temp world 300.0 310.0 320.0 ${Tfinal}
variable x universe 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
variable x uloop 15

```

### Description:

This command assigns one or more strings to a variable name for evaluation later in the input script or during a simulation.

Variables can be used in several ways in ChemCell. 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 ChemCell 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 ChemCell 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.ChemCell.variable" and "tmp.ChemCell.variable.lock" which you will see in your directory during such a ChemCell run.

The *uloop* style is identical to the *universe* style except that the strings are the integers from 1 to N. This allows generation of long list of runs (e.g. 1000) without having to list N strings in the input script.

---

For the *equal* style, a single string is specified which represents a formula that will be evaluated afresh each time the variable is used. If you want spaces in the string, enclose it in double quotes so the parser will treat it as a

single argument. For *equal* style variables the formula computes a scalar quantity, which becomes the value of the variable whenever it is evaluated.

Note that *equal* variables can produce different values at different stages of the input script or at different times during a run.

The next command cannot be used with *equal* style variables, since there is only one string.

The formula for an *equal* variable can contain a variety of quantities. The syntax for each kind of quantity is simple, but multiple quantities can be nested and combined in various ways to build up formulas of arbitrary complexity. For example, this is a valid (though strange) variable formula:

```
variable x equal "2.0 + v_MyTemp / pow(v_Volume,1/3)"
```

Specifically, an formula can contain numbers, math operations, and references to other variables.

|                 |   |
|-----------------|---|
| Number          | 0.2, 100, 1.0e20, -15.4, etc  |
| Math operations | (), -x, x+y, x-y, x*y, x/y, x^y, sqrt(x), exp(x), ln(x), log(x), sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), ceil(x), floor(x), round(x) |
| Other variables | v_abc, v_n  |

Math operations are written in the usual way, where the "x" and "y" in the examples above can be another section of the formula. Operators are evaluated left to right and have the usual precedence: unary minus before exponentiation ("^"), exponentiation before multiplication and division, and multiplication and division before addition and subtraction. Parenthesis can be used to group one or more portions of a formula and enforce a desired order of operations. Additional math operations can be specified as keywords followed by a parenthesized argument, e.g. sqrt(v\_ke). Note that ln() is the natural log; log() is the base 10 log. The ceil(), floor(), and round() operations are those in the C math library. Ceil() is the smallest integer not less than its argument. Floor() if the largest integer not greater than its argument. Round() is the nearest integer to its argument.

The current values of other variables can be accessed by prepending a "v\_" to the variable name. This will cause that variable to be evaluated.

**IMPORTANT NOTE:** If you define variables in circular manner like this:

```
variable a equal v_b
variable b equal v_a
print $a
```

then ChemCell 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 simulation domain (liters)

### Examples:

volume 8.0e-15

### Description:

Set the volume of the system for a non-spatial simulation. This will be the well-mixed volume within which reactions take place, e.g. for a stochastic Gillespie or continuum ODE simulation.

### Restrictions:

This command can only be used for non-spatial simulations. See the [global](#) command for spatial simulations.

### Related commands:

[global](#)

**Default:** none

## write\_restart command

### Syntax:

```
write_restart file
```

- file = name of file to write restart information to

### Examples:

```
write_restart restart.equil
```

### Description:

Write a binary restart file of the current state of the simulation. Only particle information is stored in a restart file.

During a long simulation, the [restart](#) command is typically used to dump restart files periodically. The `write_restart` command is useful between simulations or whenever you wish to write out a single current restart file.

Restart files can be read by a [read\\_restart](#) command to restart a simulation from a particular state. Because the file is binary (to enable exact restarts), it may not be readable on another machine.

### Restrictions:

This command can only be used for spatial simulations.

### Related commands:

[restart](#), [read\\_restart](#)

**Default:** none