Smoldyn User's Manual

for Smoldyn version 2.38, © October 2015

Steve Andrews

1. Overview of Smoldyn

Smoldyn is a computer program for simulating chemical processes on a microscopic size scale. The size scale is sufficiently detailed that all molecules of interest are simulated individually, while solvent molecules and any molecules that are not of immediate interest are only treated implicitly. In the simulation, molecules diffuse, react, are confined by surfaces, and bind to membranes, much as they would in a real chemical system.

In Smoldyn, each molecule is represented by a single point in 1-, 2-, or 3-dimensional continuous space. Simulated molecules do not have volume (typically), spatial orientations, or momenta. Because of these approximations, simulations are often accurate on spatial scales that are as small as about a nanometer and timescales down to about a microsecond. This accuracy comes at the cost of high computational intensity. For systems that are larger than tens of microns, or dynamics that unfold over tens of minutes, simulation methods that are more computationally efficient but less accurate are likely to be preferable.

Several new algorithms needed to be developed for Smoldyn to achieve the accuracy that it does, without making it so computationally demanding that it would run too slowly to be useful. In particular, the algorithm for bimolecular reactions was an early and essential component of Smoldyn (Andrews and Bray, *Physical Biology* 1:137, 2004). It is based on a theory of diffusion-limited chemical reaction rates that was derived by von Smoluchowski in 1917. The name "Smoldyn" is short for Smoluchowski dynamics, which comes from this theory.

The input to Smoldyn is a plain text configuration file. This file specifies all of the details of the system, such as the shapes and positions of membranes, the initial types and locations of molecules, diffusion coefficients, instructions for the graphical output, and so on. Smoldyn reads this file, prints out some information about the system so the user can verify that the file was written correctly, and then runs the simulation. As the simulation runs, the state of the system can be displayed to a graphics window to allow the user to watch what is happening, or to capture the simulation result in a movie. Also, it is possible to enter commands in the configuration file that are executed at runtime, and which output quantitative results from the simulation to text files. Smoldyn quits when the simulation is complete.

History of Smoldyn

When I started a post-doc in Dennis Bray's lab at the University of Cambridge in 2001, my project was to model *E. coli* bacterial chemotaxis using the MCell program. However, I wasn't eager to use MCell because of its closed source code, inability to model reactions in solution (at the time), and mathematical foundation that I didn't agree with. At least as importantly, and not MCell's fault at all, MCell only ran on Linux systems and I knew very little Linux at the time. I decided that it would be easier to write my own small simple simulator than it would be to get MCell working as I needed. I was very naive.

Before long, I discovered that I couldn't simulate bimolecular reactions in solution until I derived the mathematics for it, which ended up being several months of work and

a research paper (Andrews and Bray, 2004). Also, I found that I couldn't use or debug my small simple simulator without a text input parser and some graphical output. Thus, the nascent Smoldyn program rapidly became neither small nor simple. A year and a half later, I released a rudimentary version of Smoldyn, and I had done some other research as well, but I still hadn't gotten around to looking at *E. coli* chemotaxis. That ended up being done by my successor in Dennis's lab, Karen Lipkow.

During my second post-doc, in Adam Arkin's lab at Lawrence Berkeley National Laboratory, I continued developing Smoldyn as I found time for it, and as the few users (mostly Karen) requested additions. During this period, I designed and implemented surfaces, although I never quite managed to get them to not leak molecules. It was not until the fall of 2007, when I left Adam's lab and spent 4 months in Upi Bhalla's lab at the National Centre for Biological Sciences in Bangalore, that I finally found time to fix all the surface problems and add several other major components. Also, Upi merged Smoldyn into his MOOSE program and I started a collaboration with the Computer Research Laboratories in Pune (India) to parallelize Smoldyn, making this an extremely productive period. Unfortunately, these parallelization efforts did not end up being useful. I cleaned up the new Smoldyn additions, overhauled other portions of the code, and wrote much of the manual during my first couple of months at my next job, at the Molecular Sciences Institute, in Berkeley, California.

The next major Smoldyn additions occurred during the following year, from mid-2008 to mid-2009. I developed and implemented accurate algorithms for simulating adsorption, desorption, and partial transmission (*Physical Biology*, 2009). Meanwhile, Nathan Addy linked his libmoleculizer library to Smoldyn so that Smoldyn could automatically generate species lists and reaction networks for multimeric molecular complexes. Nathan also improved the Smoldyn build system and parallelized Smoldyn with POSIX threads. Starting in the fall of 2009, I have been a research associate in Roger Brent's lab at the Fred Hutchinson Cancer Research Center, where I've spent much of my time wrapping up the loose ends with the recent additions to Smoldyn, writing papers about it, and applying for funding for continuing work.

2. Installing Smoldyn

Smoldyn is relatively easy to download and install. Despite that, systems differ and things can go wrong, so the directions given here are quite thorough. They aren't meant to be intimidating, just helpful in case things aren't as easy as they ought to be.

2.1 System requirements

Smoldyn runs on Mac OS X, Linux, or Windows systems. Smoldyn should be able to run on any computer or version of these operating systems that is less than about 5 years old. However, Smoldyn is a computationally intensive program, so faster hardware is definitely preferable.

2.15 Pre-compiled versions

The Smoldyn distribution system changed in version 2.24. Now, there are three downloads. One has pre-compiled code for Macs, the second has pre-compiled code for Windows, and the third is the full distribution with source code and no pre-compiled code. The first two should be very easy to install. For Macs, follow the instructions in the README.txt file, which basically say to type "sudo ./install.sh" and that everything will be installed for you. For Windows, the Smoldyn program is in the bin directory, and the dll directory includes several dlls that you might need. These dlls should go in your System32 folder if you have a 32 bit version of Windows and in your SysWOW64 folder if you have a 64 bit version of Windows.

Both downloads include Smoldyn, the Smoldyn utility programs (SmolCrowd and wrl2smol), and the Libsmoldyn library, along with example files, and documentation. Libsmoldyn is a powerful new feature, but is still very much in development.

2.2 The download package

Smoldyn is available for download from the web site www.smoldyn.org. At this web site, follow the download link, and then choose the package that is most appropriate for your system. Decompressing the package is usually straightforward (double click for Mac or Windows, or use the "tar" command for Linux). The result will be a directory with a lot of files and a few subdirectories. Most of the files are used by the software build system, and are thus not important to most users. However, the following ones are worth knowing about:

cmake (directory)	empty directory for building Smoldyn
CMakeLists.txt	CMake instructions for configuring build
documentation (directory)	complete Smoldyn documentation
doc.../Smoldyn_doc1.pdf	this Smoldyn users manual
doc.../SmolCrowd_doc.pdf	manual for SmolCrowd utility
doc.../wrl2smol_doc.pdf	manual for wrl2smol utility
examples (directory)	example Smoldyn configuration files

source (directory) all of the Smoldyn source code

2.3 Compiling on Macintosh

Smoldyn requires the Unix-style interface that underlies the OS X Macintosh operating system. To access it, open the Terminal application, which should be in the Utilities subdirectory of your Applications folder.

(1) You will need a C compiler and the Make utility. To check if you have them, simply type "gcc" at a shell prompt. If it says "command not found", then you need to get it. To get it, go to http://developer.apple.com/xcode and click on the "view in Mac App store button" to be taken to the Xcode site in the Mac App store. Then, click on the "Free" button, register for a free Apple Developer Connection account if you don't have one already, and click on the same button, which is now called "Install App". This will install XCode. However, it still won't work properly. Next, start XCode and go to the "Preferences..." menu item, click on "downloads" and install the "Command line tools".

(2) OpenGL should already be installed on your computer. To check, type "`ls /System/Library/Frameworks`" and you should see folders called GLUT.framework and OpenGL.framework. If they aren't there, then you'll need to get them.

(3) You will need the CMake configuration software. To see if you already have it, type "`cmake`"; this will produce the help information if you have it, or an error message if not. If you don't have it, you need to download and install it.

(4) Libtiff is a library that Smoldyn uses for saving tiff format images, which you probably do not have. It is not required for Smoldyn to run, but it necessary to save images. One way to install Libtiff is to download it from http://www.libtiff.org, uncompress it, and install it. To install it, start a terminal window, change to the libtiff directory, and follow the README instructions: type "`./configure`", then "`make`", then "`sudo make install`" and your password. This will install libtiff header files to /usr/local/include and libtiff library archives in /usr/local/lib.

Another method (but one which I think is harder) is to use MacPorts or Fink. For MacPorts, type "`port search libtiff`". If you get the error message "`port: command not found`", then you don't have MacPorts. If this is the case, then you can get MacPorts from www.macports.org and try again. When the command works, it should list a few packages, one of which is called "`tiff @3.8.2 (graphics)`", or something very similar. Install it by typing "`sudo port install tiff`", followed by your password. This will install libtiff to /opt/local/var/macports/software/. This is great, except that the Smoldyn build system prefers for libtiff to be in /usr/local/lib. The solution is to set `LIBTIFF_CFLAGS` and `LIBTIFF_LDFLAGS` manually when you type `./configure` for Smoldyn. This will override Smoldyn's search for the libraries and will link them in properly. For Fink, exactly the same advice applies, except that Fink installs libraries to /sw. For example, if libtiff is installed to /sw/local, then configure with: "`LIBTIFF_CFLAGS="-I/sw/local/include" LIBTIFF_LDFLAGS="-L/sw/local/lib -ltiff" ./configure`".

(4) Install Smoldyn by changing to the "cmake" directory. Then type "`cmake ..`", then "`make`", and then "`sudo make install`", and finally your password. Some custom installation options can be selected with the ./configure line if you want them; they are listed below in the sections on installing to a custom location and on installation problems, and also in the Smoldyn programmers manual. To clean up temporary files, *which is essential if you want to try building a second time*, first enter "`pwd`" and confirm that you are still in the "cmake/" directory (don't continue if not!). Then, type "`rm -r *`" to clear out all prior build stuff.

(5) Test Smoldyn

(a) Type "`smoldyn -V`" to just print out the Smoldyn version number. If it doesn't work, then the most likely problem is that your system is not set up to run programs that are in your /usr/local/bin directory, which is where Smoldyn is installed by default. To fix this temporarily, type "`export PATH=$PATH:/usr/local/bin`"; to fix it permanently, although it will only take effect after you open a new terminal window, use emacs or some other editor to edit the file ~/.profile and add the line "`export PATH=$PATH:/usr/local/bin`".

(b) Type "`smoldyn examples/S8_reactions/lotvolt/lotvolt.txt`" to run a Lotka-Volterra simulation. If a graphics window doesn't appear, then the OpenGL linking somehow failed. Otherwise, press 'T' (upper-case) at some point during the simulation to save a tiff-format image of the graphical display. If it works, it will be saved to the current directory as OpenGL001.tif; if not, then the libtiff linking somehow failed.

2.4 Compiling options

Various building options are possible with the CMake build system, of which the most important are as follows. In all cases, append these to the "`cmake ..`" command.

`-DOPTION_STATIC=ON`	Build using static libraries
`-DCMAKE_BUILD_TYPE=...`	Choose CMake build type
options are: `None`, `Debug`, `Release` (default), `RelWithDebInfo`, and `MinSizeRel`	
`-DOPTION_USE_OPENGL=OFF`	Build without graphics support
`-DOPTION_USE_LIBTIFF=OFF`	Build without LibTiff support
`-DOPTION_USE_ZLIB=OFF`	Build without ZLib support
`-OPTION_TARGET_SMOLDYN=OFF`	Don't build stand-alone Smoldyn program
`-DOPTION_TARGET_LIBSMOLDYN=ON`	Build LibSmoldyn library
`-DOPTION_NSV=ON`	Build with next subvolume support
`-DOPTION_VTK=ON`	Build with VTK output support

By default, the Smoldyn build system installs Smoldyn to either the /usr or the /usr/local directories, depending on your system. These are the standard places for programs like Smoldyn, but you will need root access for the installation (typically only system administrators have the neccessary su or sudo access to install to these locations).

If you use a computer on a shared computer, you may not have this access. If this is the case, then you will have to pick a different install directory, such as ~/usr. There are standard options to configure Smoldyn to install here, for the CMake build system

The drawback to installing in a non-standard location is that your system may not find Smoldyn when you try to run it. To solve this, you need to add the directory "~/usr", or wherever you installed Smoldyn, to your PATH variable. This is explained above in instruction 5a for the regular Macintosh installation, except that here you would add "`export PATH=$PATH:~/usr/bin`".

2.5 Compiling on a UNIX/Linux system

For the most part, installing on a UNIX or Linux system is the same as for Macintosh, described above. Following are a few Linux-specific notes.

To download Smoldyn from a command line, type "`wget http://www.smoldyn.org/smoldyn-2.xx.tar.gz`", where the "xx" is the current version number. Then unpack it with "`tar xzvf smoldyn-2.xx.tar.gz`".

For a full installation, you will need OpenGL and Libtiff. I don't know how to install them for all systems, but it turned out to be easy for my Fedora release 7. I already had OpenGL, but not the OpenGL glut library nor Libtiff. To install them, I entered "`sudo yum install freeglut-devel`" and "`sudo yum install libtiff`", respectively, along with my password.

Ubuntu systems are slightly more finicky than others. First, you may need to install several things as follows. Install a C++ compiler with "`sudo apt-get install g++`", install a Python header file with "`sudo apt-get install python-dev`", install the OpenGL glut library with "`sudo apt-get install freeglut3-dev`", and install the libtiff library with "`sudo apt-get install libtiff4-dev`".

2.6 Installing on Windows

There are several pre-compiled versions of Smoldyn for Windows in the Windows directory of the download package, called smoldyn#.##.exe, where the # signs give the version number. Several versions are included because I have had a hard time getting Smoldyn to run on Windows, so one might work while others don't.

Also in the Windows directory are several dll files (Windows dynamic linked libraries), which you will probably need to install on your system to make Smoldyn work. They are: glut32.dll, libtiff3.dll, jpeg62.dll, and zlib1.dll. At one point, I also needed opengl32.dll, opengl.dll, hfxclasses45.dll, and ipl.dll, although I haven't needed them lately. Free dll files can be found easily with a little web browsing. These dlls should go in your C:\windows\System32 folder if you have a 32 bit version of Windows and in your SysWOW64 folder if you have a 64 bit version of Windows.

2.7 Line termination characters

Macintosh, Windows, and Linux all use different line termination characters. Macintosh uses a carriage return, Linux uses a line feed, and Windows uses both. These can cause problems both with the C language source files, if you're compiling yourself, and also with the configuration files that Smoldyn reads. Some compilers convert text automatically, but not all do (for example, Linux almost never does). To convert a file from Macintosh to Linux, enter the following line at a command prompt:

```
perl -pi -e 's/\r/\n/g' filename
```

To convert from Windows to Linux, enter:

```
perl -pi -e 's/\r//g' filename
```

While it shouldn't be necessary, the following line converts all Smoldyn example files from Macintosh to Linux termination characters:

```
for filename in $(find examples/* -name *.txt); do perl -pi -e 's/\r/\n/g'
$filename; done
```

2.8 Installation problems

While Smoldyn is supposed to install reasonably easily on a wide variety of systems, this has proven more difficult than we expected. Thus, if you have problems installing Smoldyn, I suggest re-reading the appropriate sections above. If these don't solve the problem, then please let me know (steven.s.andrews@gmail.com) so I can try to help you solve it, and so I can help others with similar problems.

If you manage to solve your problem, or have suggestions that might help other users, please tell me about them so I can improve the installation.

2.9 Running Smoldyn remotely

It can be helpful to have Smoldyn installed on computer A and run from computer B. Running Smoldyn without graphics is trivial. Just ssh into computer A as normal, and run Smoldyn with "`smoldyn filename.txt -t`", where the -t flag implies text-only operation. If you want graphics though, then log in with "`ssh -Y me@compA/directory`" and run Smoldyn as normal. Graphics will be slow but should be functional.

Alternatively, I've found the free software TeamViewer to be a convenient method for working on computers remotely. An advantage of this method is that it works even if there are institutional firewalls that prohibit remote computer access.

3. Using Smoldyn

3.1 Running Smoldyn

The bounce3.txt configuration file is a good one to start with because it is quite simple. It is in the examples folder, which is supplied with the Smoldyn distribution, in a sub-folder called S1_intro.

If you are working on a Linux or Linux-like system, you can run Smoldyn from a command line interface. In this case, you can run the program by just entering `smoldyn`. The program should start, display a greeting, and ask for the name of the configuration file. Enter the configuration file name, including path information. For example, if bounce3.txt is in the working directory, just enter `bounce3.txt`; if the working directory is the one with the examples folder, enter `examples/S1_intro/bounce3.txt`; an absolute path name can be used as well. Next, Smoldyn asks for the runtime flags, which are listed in table 3.1. Any combination of flags may be used, and in any order. It is also possible to enter the configuration file name and any runtime flags on the command line. For example, Smoldyn could be run and would display the parameters for bounce3.txt with

```
smoldyn examples/S1_intro/bounce3.txt -p
```

Table 3.1.1: Runtime flags

command line	Smoldyn query	result
	-	normal: parameters displayed and simulation run
-o	o	suppress output: text output files are not opened
-p	p	parameters only: simulation is not run
-q	q	quiet: parameters are not displayed
-t	t	text only: no graphics are displayed
-V	V	display version number and quit
-v	v	verbose: extra parameter information is displayed
-w	w	suppress warnings: no warnings are shown

Without a command line interface, you probably cannot specify the configuration file name or runtime flags during program startup. Instead, these need to be entered in response to the queries from Smoldyn. Macintosh applications (not run from the terminal command line) use path information that is denoted with colons:

```
Enter name of configuration file: :examples:S1_intro:bounce3.txt
```

Windows path information is entered with backslashes:

```
Enter name of configuration file: examples\S1_intro\bounce3.txt
```

Assuming Smoldyn runs at all, the most common problem that can occur at this point is that the configuration file cannot be read because it has the wrong format. If this happens, there is no choice but to fix the line termination characters, as described above.

The bounce3.txt file results in simple red and green molecules that bounce around the simulation volume for a little while (see figure). On completion, the program displays some statistics about the simulation and then terminates. While the simulation is running, and after it finishes, it is possible to press the arrow keys to rotate the graphics and get a better view of what is happening. When finished, press command-q, or select "Quit" from the file menu, to quit the program.

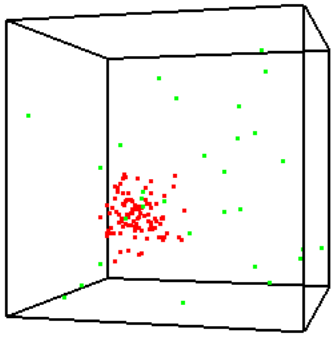

Figure 3.1.1: Graphical output from bounce3.txt.

Smoldyn can also run with 1- or 2-dimensional systems. Simple example files for these low dimension systems are bounce1.txt and bounce2.txt, which produce the outputs shown below.

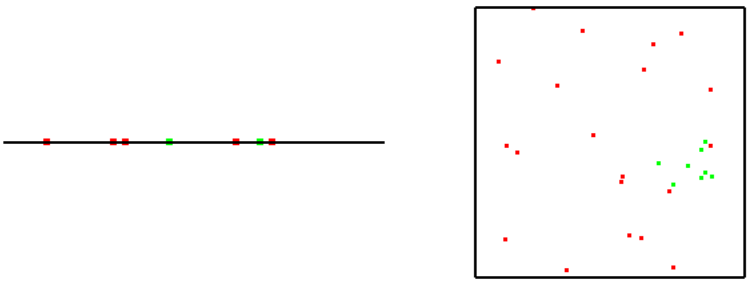

Figure 3.1.2: Graphical output from bounce1.txt and bounce2.txt, respectively.

For simulations that are run from the command line, it is possible to declare macro text substitutions using `--define`, followed by a key and replacement text. This topic is discussed in the following section, on the configuration file.

3.2 The configuration file

Configuration file basics

Configuration files, such as bounce3.txt, are simple text files. The format is a series of text lines, each of which needs to be less than 256 characters long. On each line of input, the first word describes which parameters are being set, while the rest of the line lists those parameters, separated by spaces. If Smoldyn encounters a problem with a line, it displays an error message and terminates. Possible problems include missing parameters, illegal parameter values, too many parameters, unrecognized molecule, surface, or reaction names, unrecognized statements, or others. Following is the complete bounce3.txt configuration file:

```
# Simple bouncing molecules in a 3-D system

graphics opengl

dim 3

boundaries x 0 100 r
boundaries y 0 100 r
boundaries z 0 100 r

species red green

difc red 3
difc green 1

color red red
color green green

time_start 0
time_stop 100
time_step 0.01

mol 100 red 20 30 20
mol 30 green u u u

end_file
```

In most cases, statements may be entered in any order, although some are required to be listed after others. The required sequence is not always obvious, so it is usually easiest to just try what seems most reasonable and then fix any errors that Smoldyn reports. Also, a few instructions can only be entered once, whereas others can be entered multiple times. If a parameter is entered more than once, the latter value overwrites the prior one. Parameters that are not defined in the configuration file are assigned default values.

Before discussing the configuration file statements in detail, it is helpful to skim through the bounce3.txt file quickly to get a sense of the types of things that are defined in configuration files. After an initial comment, the configuration file states that OpenGL

should used for graphics and that system is 3-dimensional. The molecule species names are called “red” and “green”. The diffusion coefficients of the red and green molecules are set and their colors are defined. The simulation starting, stopping, and stepping times are set, as are the boundaries of space. Finally, 100 red molecules and 30 green molecules are created, from the 200 total molecules that were allocated, and are placed in the simulation volume. The file ends with `end_file`.

Units

What is not in the bounce3.txt configuration file are any statements about units. In fact, no portions of Smoldyn ever use any units. Instead, it is up to the user to decide what set of units to use and to stay consistent with them. For example, two classic systems of units are cgs, which stands for centimeter-gram-second, and mks, which stands for meter-kilogram-second. Both of these are too large-scale to be convenient for most Smoldyn simulations. Instead, two systems that tend to be most useful are micron-millisecond and nanometer-microsecond. Another system that has been used successfully is to define a “pixel” as 10 nm, and to then convert all length units into pixels.

Table 3.2.1: Unit conversion

	Concentration	Diffusion coefficient	Unimolec. reactions	Bimolecular reactions	Adsorption rates
Typical value	10 μM	10 $\mu m^2 s^{-1}$	1 s^{-1}	10^5 $M^{-1}s^{-1}$	1 μm s^{-1}
mks	$6x10^{21}$ m^{-3}	10^{-11} m^2s^{-1}	1 s^{-1}	10^2 $m^3mol^{-1}s^{-1}$ $1.7x10^{-22}$ m^3s^{-1}	10^{-6} m s^{-1}
cgs	$6x10^{15}$ cm^{-3}	10^{-7} cm^2s^{-1}	1 s^{-1}	$1.7x10^{-16}$ cm^3s^{-1}	10^{-4} cm s^{-1}
μm-ms	6000 μm^{-3}	10^{-2} μm^2ms^{-1}	10^{-3} ms^{-1}	$1.7x10^{-7}$ μm^3ms^{-1}	10^{-3} μm ms^{-1}
μm-s	6000 μm^{-3}	10 μm^2s^{-1}	1 s^{-1}	$1.7x10^{-4}$ μm^3s^{-1}	1 μm s^{-1}
nm-ms	$6x10^{-6}$ nm^{-3}	10^4 nm^2ms^{-1}	10^{-3} ms^{-1}	170 nm^3ms^{-1}	1 nm ms^{-1}
nm-μs	$6x10^{-6}$ nm^{-3}	10 $nm^2\mu s^{-1}$	10^{-6} μs^{-1}	0.17 $nm^3\mu s^{-1}$	10^{-3} nm μs^{-1}
px-ms	$6x10^{-3}$ px^{-3}	100 px^2ms^{-1}	10^{-3} ms^{-1}	0.17 px^3ms^{-1}	0.1 px ms^{-1}

Notes: A pixel, abbreviated px, is defined as a length of 10 nm. In the concentration column, ‘6’ is short for 6.022045. In the bimolecular reactions column, 1.7 is short for 1.660565.

Statements about the configuration file

A few statements control the reading of the configuration file, which are now described in more detail. The first, shown in the first line of bounce3.txt, is a comment. A # symbol indicates that the remainder of the line should be ignored, whether it is the whole line as it is in bounce3.txt or just the end of the line. It is also possible to comment out entire blocks of the configuration file using `/*` to start a block-comment and `*/` to end it. For these, the `/*` or `*/` symbol combinations are each required to be at the beginning

of configuration file lines. The remainder of those lines is ignored, along with any lines between them.

It is possible to separate configuration files into multiple text files. This is done with the statement `read_file`, which simply instructs Smoldyn to continue reading from some other file until that one ends with `end_file`, which is followed by more reading of the original file. The `read_file` statement may be used anywhere in the configuration file, including within reaction definition and surface definition blocks (described below) and within files that were themselves called with a `read_file` statement. The configuration file examples/S2_config/config.txt illustrates these statements.

Smoldyn supports a rudimentary version of variables in configuration files through the `define` statements. These perform simple text substitution, much like the pre-processor `define` statement in the C programming language. As a typical example, suppose you want to run a configuration file with many different system sizes. Rather than changing each and every relevant boundary and surface definition number in the file for each run, they can all be set to the same macro key, and thus changed at once with a single `define` statement. One definition is set automatically: `FILEROOT` is replaced by the current file name, without path information and without any text that follows a '.'. Prior definitions are overwritten with new ones without causing errors or warnings. These definitions have local scope, meaning that they only lead to text replacement within the current configuration file, and not to those that it reads with `read_file`. To create a definition with broader scope, use `define_global`; the scope of these definitions is throughout the current configuration file, as well as any file or sub-file that is called by the current file. A configuration file that calls the current one is not affected by a `define_global`. To remove a definition, or all definitions, use `undefine`.

`define` statements can also be used for conditional configuration file reading. In this case, a definition is made as usual, although there is no need to specify any substitution text. Later on in the file, the `ifdefine`, `else`, and `endif` statements lead to reading of different portions of file, depending on whether the definition was made or not. A variant of the `ifdefine` statement is the `ifundefine` statement. These conditional statements should work as expected if they are used in a normal sort of manner (see any programming book for basic conditional syntax), which includes support for nested "if" statements. They can also be used successfully with highly abnormal syntaxes (for example, an `else` toggles reading on or off, regardless of the presence of any preceding `ifdefine` or `ifundefine`), although this use is discouraged since it will lead to confusing configuration files, as well as files that may not be compatible with future Smoldyn releases.

Text substitution can also be directed from the command line. If you include the command line option `--define`, followed by text of the form *key=replacement* (do not include spaces, although if you want spaces within the replacement text, then enclose it in double quotes), this is equivalent to declaring text substitution using the `define_global` statement within a configuration file. For example, to the file cmdlinedefine.txt includes the macro key "`RDIFC`" but does not define it. To run this file, define the macro key on the command line like

```
smoldyn examples/S2_config/cmdlinedefine.txt --define RDIFC=5
```

This feature simplifies running multiple simulations through a shell script. Essentially any number of definitions can be made this way. If the same key text is defined both on the command line and in the configuration file, the former takes priority.

Table 3.2.2: statements about the configuration file

`#`	single-line comment
`/* ... */`	multi-line comment
`read_file` *filename*	read *filename*, and then return
`end_file`	end of this file
`define` *key substitution*	local macro replacement text
`define_global` *key substitution*	global macro replacement text
`undefine` *key*	undefine a macro substitution
`ifdefine` *key*	start of conditional reading
`ifundefine` *key*	start of conditional reading
`else`	else condition for conditional reading
`endif`	ends conditional reading

Running multiple simulations using scripts

It is often useful to simulations over and over again, whether to collect statistics, to look for rare events, or to scan over parameter ranges. This is easily accomplished by writing a short Python script, or a script in some other high level language such as R, MatLab, Mathematica, etc. The following Python script is at `S2_config/pyscript.py`. It runs the file paramscan.txt several times using different parameter values, with results sent to the standard output and also saved to different files.

```
# A python script for scanning a parameter
import os

simnum=0
for rxnrate in [0.01,0.02,0.05,0.1,0.2,0.5,1]:
   simnum+=1
   string='smoldyn paramscan.txt --define RXNRATE=%f --define SIMNUM=%i -tqw'
              %(rxnrate,simnum)
   print(string)
   os.system(string)
```

Run this script by entering “python pyscript.txt”.

Although not described here in depth, another method for running batches of simulations is for your script to generate a Smoldyn-readable text file with the appropriate parameters, say with the file name `myparams.txt`. Then, in your master Smoldyn file, which might also be called from the same script, it includes the line “`read_file myparams.txt`”, which reads in the necessary parameters.

3.3 Space and time

Space

Smoldyn simulations can be run in a system that is 1, 2, or 3-dimensional. These can be useful for accurate simulations of systems that naturally have these dimensions. For example, a 2-dimensional system can be useful for investigating diffusional dynamics and interactions of transmembrane proteins. Smoldyn does not permit 4 or more dimensional systems because it is not clear that they would be useful and because they have not been tested. The system dimensionality is defined with the `dim` statement, which needs to be one of the first statements in a configuration file.

Along with the system dimensionality, it is necessary to specify the outermost boundaries of the system. In most cases, it is best to design the simulation so that all molecules stay within the system boundaries, although this is not required. All simulation processes are performed outside of the system boundaries exactly as they are within the boundaries. Boundaries are used by Smoldyn to allow efficient simulation and for scaling the graphical display. They are typically defined with the `boundaries` statement, as seen in the example S1_intro/bounce3.txt. Boundaries may be reflective, transparent, absorbing, or periodic. Reflective means that all molecules that diffuse into a boundary will be reflected back into the system. Transparent, which is the default type, means that molecules just diffuse through the boundary as though it weren't there. With absorbing boundaries, any molecule that touches a boundary is immediately removed from the system. Finally, with periodic boundaries, which are also called wrap-around or toroidal boundaries, any molecule that diffuses off of one side of space is instantly moved to the opposite edge of space; these are useful for simulating a small portion of a large system while avoiding edge effects.

On rare occasion, it might be desirable to have asymmetric system boundary types. For example, one side of a system might be reflective while the other is absorbing. To accomplish this, use the `low_wall` and `high_wall` statements instead of a `boundary` statement. This is illustrated in the example file S3_space/bounds1.txt.

These boundaries of the entire system are different from surfaces, which are described below. However, they have enough in common that Smoldyn does not work well with both at once. Thus, *if any surfaces are used, the system boundaries will always behave as though the types are transparent, whether they are defined that way or not*. Thus, if there are surfaces, it is usually best to use the boundaries statement without a type parameter, which will lead to the default transparent type. To account for the transparent boundaries, an outside surface may be needed that keeps molecules within the system. The one exception to these suggestions arises for systems with both surfaces and periodic boundary conditions. To accomplish this with the maximum accuracy, set the boundary types to periodic (although they will behave as though they are transparent) and create jump type surfaces, described below, at each outside edge that send molecules to the far sides. The reason for specifying that the boundaries are periodic is that they will then allow bimolecular reactions that occur with one molecule on each side of the system. This will probably yield a negligible improvement in results, but nevertheless removes a potential artifact. This is illustrated in the example S3_space/bounds2.txt.

Time

A simulation runs for a fixed amount of simulated time, using constant length time steps. The simulation starting time is set with `time_start` and the stopping time is set with `time_stop`. For simulations that are interrupted and then continued, the `time_now` statement allows the initial time to be set to a value that is intermediate between the starting and stopping times.

The size of the time step is set easily enough with `time_step`, although knowing what value to use is an art. Smoldyn always becomes more accurate, and runs more slowly, as shorter time steps are used. Thus, an important rule for picking a time step size is to compare the results that are produced for one value with those produced with a time step that is half as long; if the results are identical, within stochastic noise, then the longer time step value is adequate. If not, then a smaller time step needs to be used.

As an initial guess for what time step to use, time steps can be chosen from the spatial resolution that is required. The average displacement of a molecule, which has diffusion coefficient D, during one time step is $s = (2D\Delta t)^{1/2}$, where Δt is the time step. Turning this around, to achieve spatial resolution of s, the time step needs to obey

$$\Delta t < \frac{s^2}{2D_{\max}}$$

where D_{max} is the diffusion coefficient of the fastest diffusing species. The overall spatial resolution for a simulation, which is the largest rms step length, is displayed in the "molecule parameters" section of the configuration file diagnostics output. For good accuracy, the spatial resolution should be significantly smaller than geometric features or than radii of curvature, for curved objects.

Other considerations for choosing the time step are the characteristic time scales of the unimolecular and bimolecular reactions. For good accuracy, the time step should generally be significantly shorter than the characteristic time scale of any reaction. Using k as the reaction rate constants, unimolecular and bimolecular reactions lead to the respective time step constraints

$$\Delta t < \frac{1}{k}$$

$$\Delta t < \frac{[\mathrm{A}]+[\mathrm{B}]}{k[\mathrm{A}][\mathrm{B}]}$$

The latter equation is for the reaction A + B → products. These values are displayed in the "reaction parameters" section of the configuration file diagnostics output. While the time scale for unimolecular reactions is independent of concentrations, the time scale for bimolecular reactions clearly depends on concentrations. Thus, the time scale that is displayed for bimolecular reactions is only a rough guide at best; it does not account for the changing concentrations of the reactants nor for local variations in concentrations.

As an initial guess, the time step that is chosen should be the smallest of those that are suggested here for all of these processes. Afterwards, it is usually worth running

several trial simulations with longer or shorter time steps to see what the longest time step is that still yields sufficiently accurate results.

Table 3.3.1: statements that define space and time

`dim` *dim*	system dimensionality: 1, 2, or 3
`boundaries` *dim pos1 pos2*	system boundaries on dimension *dim*
`boundaries` *dim pos1 pos2 type*	same, for systems without surfaces
`low_wall` *dim pos type*	specify single low-side boundary
`high_wall` *dim pos type*	specify single high-side boundary
`time_start` *time*	starting time of simulation
`time_stop` *time*	stopping time of simulation
`time_step` *time*	time step for the simulation
`time_now` *time*	current time of the simulation

Technical discussion of time steps

A major focus of the design of Smoldyn has been to make it so that results are indistinguishable from those that would be obtained if the simulated time increased continuously. This goal cannot be achieved perfectly. Instead, the algorithms are written so that the simulation approaches the Smoluchowski description of reaction-diffusion systems as the time step is reduced towards zero. Also, it maintains as much accuracy as possible for longer time steps. This topic is discussed in detail in the research paper "Stochastic simulation of chemical reactions with spatial resolution and single molecule detail" by Steven Andrews and Dennis Bray (*Physical Biology* 1:137-151, 2004).

In concept, the system is observed at a fixed time, then it evolves to some new state, then it is observed again, and so forth. This leads to the following sequence of program operations:

---------- time = t ----------
- observe and manipulate system
- graphics are drawn
- molecules diffuse
- desorption and surface-state transitions
- surface or boundary interactions
- reactions
  - 0th order reactions
  - 1st order reactions
  - 2nd order reactions
  - reaction products are added to system
- surface interactions of reaction products

---------- time = $t + \Delta t$ ----------

After commands are run, graphics are displayed to OpenGL if that is enabled. The evolution over a finite time step starts by diffusing all mobile molecules. In the process, some end up across internal surfaces or the external boundary. These are reflected, transmitted, absorbed, or transported as needed. Next, reactions are treated in a semi-synchronous fashion. They are asynchronous in that all zeroth order reactions are simulated first, then unimolecular reactions, and finally bimolecular reactions. With bimolecular reactions, if a molecule is within the binding radii of two different other molecules, then it ends up reacting with only the first one that is checked, which is arbitrary (but not necessarily random). Reactions are synchronous in that reactants are removed from the system as soon as they react and products are not added into the system until all reactions have been completed. This prevents reactants from reacting twice during a time step and it prevents products from one reaction from reacting again during the same time step. As it is possible for reactions to produce molecules that are across internal surfaces or outside the system walls, those products are then reflected back into the system. At this point, the system has fully evolved by one time step. All molecules are inside the system walls and essentially no pairs of molecules are within their binding radii (the exception is that products of a bimolecular reaction with an unbinding radius might be initially placed within the binding radius of another reactant).

Each of the individual routines that is executed during a time step exactly produces the results of the Smoluchowski description, or yields kinetics that exactly match those that were requested by the user. However, the simulation is not exact for all length time steps because it cannot exactly account for interactions between the various phenomena. For example, if a system was simulated that only had unimolecular reactions and the products of those reactions did not react, then the simulation would yield exactly correct results using any length time step. However, if the products could react, then there would be interactions between reactions and there would be small errors. In this case, the error arises because Smoldyn does not allow a molecule to be in existence for less than the length of one time step.

3.4 Molecules

About molecules

In Smoldyn, each individual molecule is represented as a separate point-like particle. These particles have no volume, so they do not collide with each other when they are simply diffusing (however, see "excluded volume reactions" in the reactions section, below, which can give molecules excluded volume). Because of the rapid collisions that occur for solvated molecules, both rotational correlations and momentum correlations damp out rapidly in most biochemical systems, so orientations and momenta are ignored in Smoldyn as well.

Each molecule has a molecular species. Enter the names for these species with the `species` statement.

Each molecule is allowed to exist in any of five states: (1) not bound to any surface (called solution state), (2) bound to the front of a surface, (3) bound to the back of a surface, (4) bound across a surface in the "up" direction, or (5) bound across a surface in the "down" direction. While the surface-bound states are intended to represent specific molecule attachments to membranes, they can also be used for other purposes; for example, you can specify that a trans-membrane protein is normally in its "up" state, but that it's in its "down" state when it is in a lipid raft.

Molecules that are not bound to surfaces are added with the `mol` statement. This is a reasonably versatile statement in that, on each axis, it allows molecules to be placed randomly within the simulation volume, randomly within some smaller region, or at a specific location. The `surface_mol` statement is used to add molecules that are bound to surfaces, although it cannot be entered in the configuration file until the appropriate surface has been set up. Similarly, `compart_mol` is used to add molecules to compartments, which are regions between surfaces, but it also cannot be entered until more things have been set up. The statements about molecules mentioned thus far, with the exception of the last two, are shown in either S1_intro/bounce3.txt or S4_molecules/molecule.txt.

Diffusion

Molecules in Smoldyn diffuse according to the diffusion coefficient that is entered for the appropriate species and state. These coefficients are entered with the `difc` statement. Although it has not proven to be particularly useful, it is also possible for Smoldyn to simulate anisotropic diffusion, meaning that molecules diffuse more rapidly in some directions than in others. Anisotropic diffusion is specified with a diffusion coefficient matrix using the `difm` statement.

Isotropic diffusion rates were tested quantitatively with the diffi.txt configuration file. In this file, all molecules start in the center of space, the boundaries are made transparent so molecules diffuse completely freely, and red, green, and blue molecules diffuse with different diffusion coefficients. Using a runtime command in the configuration file, described below, Smoldyn outputs the moments of the molecular distributions to text files. They were analyzed with the Excel file diffi.xls, which is also

in the S4_molecules folder. From this Excel file, the graphical and numerical results are shown in figure 3.4.1, along with theoretical predictions.

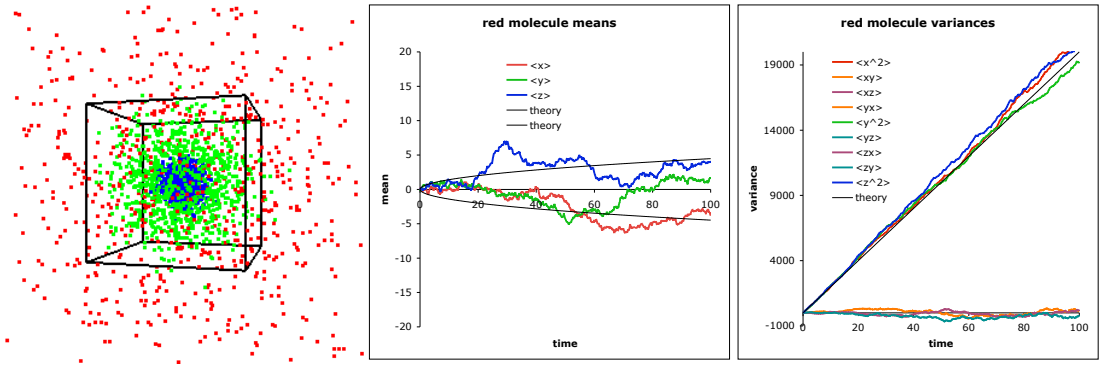

Figure 3.4.1: Graphical and numerical output from diffi.txt.

The middle panel of the figure shows that the mean position of the red molecules, on each of the three coordinates, stays near zero although with fluctuations. This is as expected for free diffusion. The expected fluctuation size, shown in the panel with light black lines, is given with

$$\left|\text{mean - starting point}\right| \approx \sqrt{\frac{2Dt}{n}}$$

where D is the diffusion coefficient, t is the simulation time, and n is the number of molecules. This equation agrees well with simulation data. The second moment of the molecule displacements is a matrix quantity which gives the variance on each pair of axes of the distribution of positions, shown in the third panel. For example, the variance matrix element for axes x and y is

$$v_{xy} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

The overbars indicate mean values for the distribution. Because diffusion on different axes is independent, the off-diagonal variances (v_{xy}, v_{xz}, and v_{yz}) are expected to be about 0, but with some fluctuations, as is seen in the figure. The diagonal variances (v_{xx}, v_{yy}, and v_{zz}) are each expected to increase as approximately

$$v_{xx} \approx v_{yy} \approx v_{zz} \approx 2Dt$$

Again, this is seen in the figure. Similar figures for the green and blue molecules, which are not presented, showed similarly good agreement between the simulation data and theory.

Anisotropic diffusion was investigated with the example file diffa.txt. In this case, the diffusion equation is

$$\dot{u} = \nabla \cdot \mathbf{D} \nabla u$$

Here, u can be interpreted as either the probability density for a single molecule or as the concentration of a macroscopic collection of molecules, and **D** is the diffusion matrix. **D** is symmetric. *The matrix that is entered in the configuration file for anisotropic diffusion, using the* `difm` *statement, is the square root of the diffusion matrix* because the square root is much more convenient for calculating expectation molecule displacements. Matrix square roots can be calculated with MatLab, Mathematica, or other methods. Note that the symmetric property of **D** implies some symmetry properties for its square root as well (for example, a symmetric square root leads to a symmetric **D**). If **D** is diagonal, the square root of the matrix is found by simply replacing each element with its square root. If **D** is equal to the identity matrix times a constant, D, the equation reduces to the standard isotropic diffusion equation. The example file diffa.txt illustrates the use of the `difm` statement; the relevant lines are

```
difm red 1 0 0 0 0 0 0 0 2
difm green 1 2 3 2 0 4 3 4 1
```

The former line leads to anisotropic diffusion of red molecules with a diffusion coefficient of 1 on the x-axis, 0 on the y-axis, and 4 on the z-axis. The latter leads to anisotropic diffusion with off-diagonal components. This matrix is interpreted to be

$$\sqrt{\mathbf{D}} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 4 \\ 3 & 4 & 1 \end{bmatrix}$$

Results are shown in figure 3.4.2.

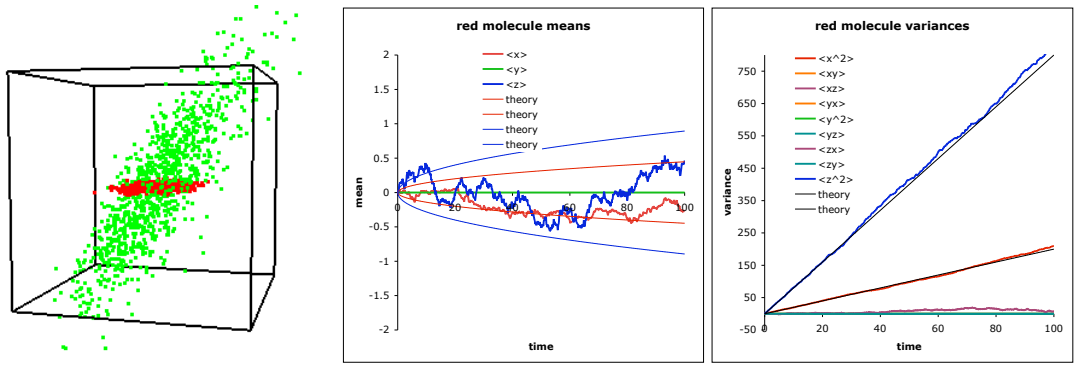

Figure 3.4.2: Graphical and numerical results from diffa.txt.

In the figure, it can be seen that the red molecules diffuse only on the *x*-*z*-plane, whereas the green molecules diffuse into an elliptical pattern that is not aligned with the axes. The red molecule data are graphed, where it is shown that *x*-values diffuse slowly, *y*-values don't diffuse at all, and *z*-values diffuse rapidly. The means and variances agree well with theory.

Drift

In addition to diffusion, molecules can drift, meaning that they move with a fixed speed and in a fixed direction. Up to version 2.26, drift could only be defined relative to the global system coordinates. For this method, which is supported in subsequent versions as well, enter the drift rate using the `drift` statement, followed by the velocity vector. Surface-bound molecules drift as well, although they are constrained to surfaces, so their actual velocity depends on the overlap of the drift vector and the surface orientation (e.g. a molecule's velocity is zero if the local surface is perpendicular to the drift vector and it equals the drift vector if that vector can lie within the the local surface orientation).

New in version 2.27, surface-bound molecules can also drift relative to the coordinates of their surface panel. Specify this with the `surface_drift` statement. For a 2-D system, surfaces are 1-D objects, so the surface-bound drift vector is a single number. It is the drift rate along "rectangles," "triangles," "spheres," etc., all of which are really just different shape lines. For a 3-D system, surfaces are 2-D objects, so the surface-bound drift vector includes two values, which generally use the most obvious orthogonal coordinates for each panel shape. For a cylinder, for example, the former number is the drift rate parallel to the cylinder axis and the latter is the drift rate around the cylinder. A possible use of surface-bound drift would be to simulate molecular motor motion along a cylinder that represents a microtubule.

Molecule lists

From a user's point of view, Smoldyn molecules follow a Western life trajectory: some chemical reaction causes a new molecule to be born from nothing, it diffuses around in space for a while, and then it undergoes a reaction and vanishes again into nothingness (or maybe goes to molecule heaven). Internally though, the situation is closer to a Wheel of Life: there are a fixed number of molecules that cycle indefinitely between "live" and "dead" states and which are assigned a new species type at each reincarnation. The dead molecule list is of no importance to the functioning of the simulation, but merely stores molecules when they are not currently active in the simulated system. The size and current population of the dead list are displayed in the molecule section of the configuration file diagnostics if you choose verbose output.

Active molecules in a simulation are stored in one or more live lists. As a default, all live molecules that diffuse, meaning that the diffusion coefficient is non-zero, are stored in a list called "diffuselist" while all fixed molecules are stored in a separate live list called "fixedlist." The separation of the molecules into these two lists speeds up the simulation because all molecules in fixedlist can be safely ignored during diffusion calculations or surface checking.

Additional live lists can be beneficial as well. For example, consider the equilibrium chemical reaction

$$A + B \leftrightarrow C$$

The only bimolecular reaction possible is between A and B molecules, so there is no need to check each and every A-A, B-B, A-C, B-C, and C-C molecule pair as well to look for more possible reactions. In this case, storing A, B, and C molecules in three separate lists means that potential A-B reactions can be checked without having to scan over all of the other combinations too. This is done in the example file S4_molecules/mollist.txt, where it is found that using three molecule lists for A, B, and C leads to a simulation that runs 30% faster than using just one molecule list. With a Michaelis-Menten reaction, the difference was found to be closer to a 4-fold improvement.

While it might seem best to have one molecule list per molecular species, it is not quite so simple. It is often the case in biology modeling that many chemical species will exist at very low copy number. In particular, a protein that can bind any of several ligands needs to be defined as separate molecular species for each possible combination of bound and unbound ligands. This number grows exponentially with the number of binding sites, leading to a problem called combinatorial explosion. Because there are so many molecular species, there are relatively few molecules of each one. Returning to the Smoldyn molecule lists, each list slows the simulation speed by a small amount. Thus, adding lists is worthwhile if each list has many molecules in it, but not if most lists are nearly empty.

At least for the present, Smoldyn does not automatically determine what set of molecule lists will lead to the most efficient simulation, so it is up to the user make his or her best guess. Molecule lists are defined with the statement `molecule_lists` and molecule species are assigned to the lists with `mol_list`. Any molecules that are unassigned with the `mol_list` statement are automatically assigned to new a list called "unassignedlist".

Enhanced wildcard support

In many statements that work with molecular species, you can specify multiple species using the wildcard characters '?' and '*', along with the logical operators '|' and '&'. A question mark can represent exactly one character and an asterisk can represent zero or more characters. For example, if you want protein Fus3 to have a different diffusion coefficient in the cytoplasm as in the nucleus, you might define it as two species, Fus3_cyto and Fu3_nucl. Then, you could specify that they are both colored red using "`color Fus3_* red`".

Wildcard logical operators are '|' for OR and '&' for AND, along with braces to enforce an order of operation. Use the former operator to enumerate a set of options. Continuing with the above example, you could specify that both species should be red with "`Fus3_{cyto|nucl}`", where this is now more specific than using the asterisk wildcard character. Use the ampersand to specify that multiple terms are in a species name but that the order of the terms is unimportant. For example, "a&b&c" represents any of the 6 species: abc, acb, bac, bca, cab, and cba. The '&' operator takes precedence

over the 'l' operator so, for example, "a|b&c" represents any of: a, bc, and cb. On the other hand, {a|b}&c represents any of: ac, bc, ca, and cb.

Wildcard characters were new in version 2.26 and are still being developed. At present, they can be used for most statements where a species property is defined (e.g. diffusion coefficient, color, surface interaction rate). They can also be used for the `text_display` statement and in a few commands (mostly those that are some variant of `molcount`). In the future, it will also be possible to define reactions using wildcard characters, leading to a simple version of rule-based reaction network generation.

Table 3.4.1: statements about molecules

`species` *name*$_1$ *name*$_2$ … *name*$_n$	names of species
`difc` *species*(*state*) *value*	diffusion coefficient
`difm` *species* (*state*) m_0 m_1 … $m_{dim*dim-1}$	diffusion matrix
`drift` *species* (*state*) v_0 v_1 … v_{dim-1}	global drift vector
`surface_drift` *species* (*state*) *surface pshape* v_0 v_1	surface-relative drift vector
`mol` *nmol species* pos_0 pos_1 … pos_{dim-1}	solution molecules placed in system
`surface_mol` *nmol species*(*state*) *surface pshape panel* [pos_0 pos_1 … pos_{dim-1}]	surface-bound molecules placed in system
`compartment_mol` *nmol species compartment*	molecules placed in compartment
`molecule_lists` *listname*$_1$ *listname*$_2$ …	names of molecule lists
`mol_list` *species*(*state*) *listname*	assignment of molecule to a list

3.5 Graphics

Graphics window display

A strong feature of Smoldyn is the real-time graphical output. Graphics are useful for designing and debugging configuration files, for understanding the results of a simulation, and for communicating simulation results to others. They have also proven invaluable for debugging Smoldyn.

Graphical output, and the overall type of graphics, is enabled with the `graphics` statement which is included at the beginning of most of the example files. Smoldyn supports the graphics options: "none", "opengl", "opengl_good", and "opengl_better". The "none" option means that no graphics are displayed, which is convenient for running batches of quantitative simulations. The "opengl" option shows molecules as small squares that don't account for which is in front of others. This is poor rendering quality but is fast to simulate. The "opengl_good" option replaces these squares with circles that are a little better looking, that account for depth-testing, and are much slower to render. Finally, the "opengl_better" option allows for the placement of light sources, for molecules to be shiny spheres, and for surfaces to be shiny. This yields fairly good quality results.

Graphical rendering can be as computationally intensive as the simulation itself, so it can be prudent to not display the system at every simulation time step, but only every *n*'th time step. This is done with the `graphic_iter` statement. Alternatively, exactly the opposite may be wanted. It may be that the simulation runs too quickly for one to understand what's being shown in the graphics window as it happens. To slow the simulation down, use the `graphic_delay` statement.

If you use the graphical output, then Smoldyn does not stop when the simulation is complete, but it instead lets you continue manipulating the graphics. When you are done, press 'Q' (shift and 'q' key). You can also stop using command-q, but that is less good because it forces Smoldyn to quit immediately rather than simply telling Smoldyn to finish its tasks (such as closing files and freeing memory) and then quit. If you want Smoldyn to stop as soon as the simulation is complete, use the `quit_at_end` statement.

The graphical display can be manipulated during the simulation using the keyboard. These keys and their actions are listed in the table shown below. Note that it is possible to rotate the system about either the viewing axes with the arrow keys, or about the object axes with the x, y, and z keys.

Table 3.5.1: Graphics manipulations during runtime

Key press	dimensions	function
space	1,2,3	toggle pause mode between on and off
Q	1,2,3	quit
T	1,2,3	save image as TIFF file
0	1,2,3	reset view to default
arrows	3	rotate object
shift + arrows	1,2,3	pan object
=	1,2,3	zoom in

-	1,2,3	zoom out
x,y,z	3	rotate counterclockwise about object axis
X,Y,Z	3	rotate clockwise about object axis

Drawing the system

Several statements control the drawing of the system. The background color is set with `background_color`, the system boundaries are drawn with the line thickness that is set with `frame_thickness` and the color that is set with `frame_color`. Although the feature is usually turned off, the `grid_thickness` and `grid_color` statements can be used to display the virtual boxes into which the system is divided (see the optimization section). Molecules are drawn with a size that is set with `display_size` and a color set with `color`. All of the statements that set colors require either color words chosen from the table below, or numbers for the red, green, and blue color channels. Regarding the molecule display size, dimensions are in pixels if the output style is just "opengl" and are in the same length units are used in the rest of the configuration file if the output style is "opengl_good".

Table 3.5.2: Available colors

maroon	olive	royal	darkred
red	green	sky	darkorange
scarlet	chartrouse	aquamarine	darkyellow
rose	khaki	violet	darkgreen
brick	purple	mauve	darkblue
pink	magenta	orchid	darkviolet
brown	fuchsia	plum	lightred
tan	lime	azure	lightorange
sienna	teal	black	lightyellow
orange	aqua	gray	lightgreen
salmon	cyan	grey	lightblue
coral	blue	silver	lightviolet
yellow	navy	slate	
gold	turquoise	white	

Some color statements also allow an alpha value, which is used for partially transparent objects; 1 is opaque and 0 is completely transparent. The OpenGL graphics library does not do a good job of supporting partially transparent objects, so alpha values between 0 and 1 often lead to mediocre rendering.

Text display

A few text items can be written to the graphics window during the simulation, all of which are displayed in the upper left corner of the graphics window. These are the

simulation time and the numbers of different molecular species in the simulation. Use the `text_color` and `text_display` statements to control this output.

TIFF files

Graphical images may be saved as TIFF images that are copied from the graphical display. Thus, the saved image size and resolution are the same as they are on the screen. A single snapshot can be saved during a simulation by pressing 'T' (uppercase). As a default it is saved as "OpenGL001.tiff", which will be in the same file folder as the configuration file. Alternatively, the configuration file statements `tiff_name` can be used to set the basic name of the file (a name of "picture" will end up being saved as "picture001.tiff"). The numerical suffix of the name can be set with `tiff_min` and `tiff_max`. The `tiff_max` value can be set to arbitrarily large numbers, although reasonable values are recommended so that vast numbers of useless tiff files can't be saved by accident.

A sequence of TIFF files can be saved automatically with the `tiff_iter` statement, allowing one to save an image sequence for later compilation into a movie. TIFF files can also be saved automatically with the `keypress T` command, which allows more versatile timing than the `tiff_iter` statement. Compiling an image sequence into a movie is easy with Apple's QuickTime Pro or with various other programs.

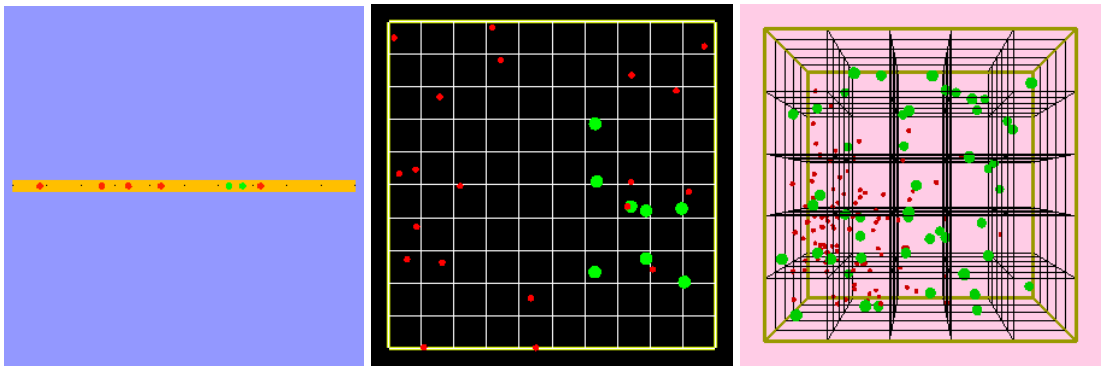

Figure 3.5.1: Graphics with 1-D, 2-D, and 3-D simulations made with the files graphics1.txt, graphics2.txt, and graphics3.txt, all from the S5_graphics directory. All of these use the graphics quality "opengl_good".

Table 3.5.1: statements about graphics (not including surfaces)

`graphics` *str*	graphical output method
`graphic_iter` *int*	time steps run between renderings
`graphic_delay` *float*	additional delay between renderings
`quit_at_end` *yes/no*	Smoldyn should quit when it's done
`frame_thickness` *int*	thickness of system frame
`frame_color` *red green blue*	color of system frame

`grid_thickness` *int*	thickness of virtual box grid
`grid_color` *red green blue*	color of virtual box grid
`background_color` *red green blue*	background color
`display_size` *name float*	size of display for a molecule species
`color` *name red green blue*	color for a molecule species
`text_color` *color*	color for text display
`text_display` *item*	item that should be displayed with text
`tiff_iter` *int*	time steps between TIFF savings
`tiff_name` *name*	root name of TIFF files
`tiff_min` *int*	initial suffix for TIFF files
`tiff_max` *int*	largest possible TIFF suffix

3.6 Runtime commands

Command basics

The design of a simulation can be broken down into two portions. One portion represents the physical system, including its boundaries, surfaces, molecules, and chemical reactions. These are the core components of Smoldyn and are simulated by the main program. The other portion represents the action of the experimenter, which include observations and manipulations of the system. As with the parameters of the physical system, these actions are also listed in the configuration file. They are listed as a series of commands and execution times.

There are no rules regarding what commands can and cannot do. Thus, in principle, commands could be used to measure any aspect of the simulated system at any time. Or, other commands could be used to manipulate any aspect of the system, regardless of whether the manipulations have any physical basis. In practice, there is a limited set of commands that have been written (listed below in the reference section) so the range of what can actually be done with commands is limited to what those in this list can do. Alternatively, a somewhat adventurous user can write his or her own source code to create a new command, as explained below. Because commands do not have to follow the rules that the rest of the code does, they are easy to add and are powerful, but they also tend to be less stable and less well optimized than the core program.

Commands are entered in a configuration file with the statement `cmd`, followed by some information about the execution timing, the specific command name, and finally any parameters for the command. Here are some examples:

```
cmd b pause
cmd e ifno ATP stop
cmd n 100 molcount outfile.txt
```

The first one instructs the simulation to pause before it starts running, the second says that the simulation should stop if there are no molecules named ATP, and the third tells Smoldyn to print a count of all molecules to the file called outfile.txt every 100 iterations. In contrast to the statements that define the physical system, runtime commands are not parsed or interpreted until the simulation time when they are supposed to be executed. When a command is executed, Smoldyn processes it with a runtime command interpreter. If there are errors in command parameters, such as a missing or nonsensical parameter, these are not caught until the command is executed during the simulation.

Command execution timing works in either of two ways. A command can be performed at real-valued simulation times, such as before the simulation starts, at some particular time, or repeatedly at fixed time intervals. Alternatively, a command can be performed after some specified number of time steps. This avoids minor timing problems that can arise from round-off error. Commands for these two methods are stored in the continuous-time and integer command queues, respectively. If two commands are entered with the exact same timing instructions, then, at each invocation, they are performed in the same order as they are listed in the configuration file. On the other hand, the order may differ if their timing instructions differ; to be precise, they are

executed in the order from the one that was least recently performed to the one that was most recently performed. If both integer and continuous time queue commands are supposed to execute at the same time step, then all of integer queue commands are performed first. Command timing is demonstrated with the configuration files S6_commands/cmdtime1.txt and S6_commands/cmdtime2.txt.

Table 3.6.1: Runtime command timing

code	parameters	execution timing
continuous time queue		
`b`		runs once, before simulation starts
`a`		runs once, after simulation ends
`@`	*time*	runs once, at ≥ *time*
`i`	*on off dt*	runs every *dt*, from ≥ *on* until ≤ *off*
`x`	*on off dt xt*	geometric progression
integer queue		
`B`		runs once, before simulation starts
`A`		runs once, after simulation ends
`&`	*i*	runs once, at iteration *i*
`I`	*oni offi dti*	runs every *dti* iteration, from ≥ *oni* to ≤ *offi*
`E`		run every time step
`N`	*n*	runs every *n* time steps

A few deprecated codes, which are in addition to the codes listed above, are that `j` is equivalent to `I`, `e` is equivalent to `E`, and `n` is equivalent to `N`.

Each command is one of three main types: control, observe, or manipulate. Control commands control the simulation operation. For example, a command called `keypress`, followed by a letter, causes the simulation to act as though that key had been pressed by the user. This can be useful for modifying the display automatically. Observation commands read information from the simulation data structures, analyze the data some, and output results to text files. The precison of numerical output values can be set using the `output_precision` statement. Neither control nor observation commands modify any aspect of the simulation. Manipulation commands modify the simulation parameters, such as the addition, removal, or replacement of molecules, or the modification of reaction rate constants. These commands do not produce any output. Yet a fourth type of command is the conditional command. These test for certain simulation conditions, such as there being more than some number of some molecule species, and then run a second command if the conditions are met. Each conditional command is characterized as being one of the three main types based on the type of its second command.

Output files

For observation commands to work, one typically needs to declare the output file names with the statements `output_files` or `append_files`. The exception to this is if

output should go to the standard output or standard error location, which are typically the terminal window. These are called "stdout" and "stderr", respectively, exactly as in C or C++. These can be declared with the `output_files` statement but don't need to be.

To save output files in a subdirectory, the subdirectory path is declared with the `output_root` statement. Note that the path needs to end with a '/', if you're working on a Linux system, or a comparable separator on other systems. This subdirectory path is concatenated on the end of the path that was used for the configuration file. It is possible to save a stack of files in which there is a separate file for each of many sequential observations. These are created with the `output_file_number` statement, which defines the starting suffix number for the file stack. Zero, which is the default, indicates no suffix number, whereas other numbers lead to a 3 digit suffix. The suffix number is incremented with the command `incrementfile`. The complete output filename is a concatenation of: the path for the configuration file, the string declared with `output_root`, the file name declared with `output_files` minus any suffix that starts with a '.', an underscore and the suffix number declared with `output_file_number`, and finally any suffix that starts with a '.'. Here is an example, using Linux path notation:

working directory: `theory`
configuration file: `theory/expt1/myconfig.txt`
desired output files: `theory/expt1/results/outfile_001.txt`
`theory/expt1/results/outfile_002.txt`
...

configuration file excerpt:

```
output_root results/
output_files outfile.txt
output_file_number outfile.txt 1
cmd n 100 incrementfile outfile.txt
cmd e molcount outfile.txt
```

starting Smoldyn: `smoldyn expt1/myconfig.txt`

There are some differences with Macintosh file formats. Instead of using a forward-slash to indicate subdirectories, a colon is used. Also, a preceding colon is required to indicate a directory rather than a file. Thus, to convert the above example to Macintosh notation, each slash is replaced by a colon and the configuration file for Smoldyn would be entered as "`:expt1:myconfig.txt`" (note the initial colon). Because of the potential for confusion with output file names, complete pathnames (relative to the working directory) are displayed at start-up with the simulation parameters.

An example that is essentially identical to the one shown above is in given in the example file S6_commands/cmdfile.txt. Upon running it and looking at the results, you will discover that the first output file, cmdfileout_001.txt, is empty, whereas all of the others are full, as expected. The empty file arises because the file number is incremented at the very beginning, before the molcount command is invoked for the first time. This could be remedied by using slightly more sophisticated command timing with the 'i' or 'j' timing codes.

Specific commands

All of the commands are listed below in the reference section, which is the definitive source of information about them. Most of the commands are also demonstrated in the example files S6_commands/cmdobserve.txt and S6_commands/cmdmanipulate.txt. Of the full list of commands, some are quite useful, some are rarely used, and some have been superceded by newer code. The last category includes several that implement rudimentary reflecting surfaces, which were written before a good treatment of surfaces was added to the core program. Of the more useful commands, a few deserve special mention.

The `molcount` command, and several variations of it, are used to save the numbers of each kind of molecule as a function of time. These are often the most useful text output commands from Smoldyn.

The `savesim` command causes the entire simulation state to be saved to a file as a configuration file that can be read by Smoldyn. With it, one can save a simulation mid-run and then continue running it later. This can be useful as a backup for intermediate results or for building starting states for complex simulations in several stages.

The `keypress` command creates an event that the program responds to, as though the user had pressed this key. For example, at the end of a simulation that uses graphics, the graphics window is left on the screen until the user selects quit from the menu or presses 'Q'. This quitting can also be programmed into the configuration file with "`cmd a keypress Q`". Arrows and other keypress options can be entered as well.

The `set` command enables you to enter essentially any configuration file statement mid-simulation. For example, the command "`set species green`" creates the species named "`green`" when the command is invoked, rather than at the beginning of the simulation. It's also possible to create surfaces, add reactions, etc. mid-simulation.

Writing your own commands

Commands are written in C, like the rest of the Smoldyn code, and are compiled with the rest of the code. Compared to the core program, commands are relatively easy to write, although they still aren't easy. In most cases, one needs to know the intricacies of the data structures in order to properly navigate or modify them. This information is all documented in part II of the manual.

Commands are allowed to look at or modify any part of the simulation data structures, making them quite powerful, but also problematic if they are written incorrectly. There is no safety system that protects the core program from commands, so writing the wrong thing to a data structure can easily cause Smoldyn to crash. For this reason, it may be easier to write observation commands than manipulation commands.

To write a command, do the following steps, which can be done in any order:

1. Write a description of the new command that will go into the reference section of the user's manual, modeling it on those that are given below.
2. In smolcmd.c, add a new declaration to the top of the file for the command, which looks like:

```
enum CMDcode cmdname(simptr sim,cmdptr cmd,char *line2);
```

3. The first function of smolcmd.c is `docommand`. In it, add an "`else if(…)`" line for the new command. It looks like:

```
else if(!strcmp(word,"name")) return cmdname(sim,cmd,line2);
```

4. Write the function for the new command, modeling it on the command functions currently in smolcmd.c. The documentation file SimCommand.doc may be helpful.
5. Proofread the function and test the command.
6. Write documentation about the command for part II of the Smoldyn manual.
7. Mention the command in the Smoldyn modifications portion of part II of the manual.

For a command to be included in updated Smoldyn releases, send the code to me at steven.s.andrews@gmail.com. Assuming I like it, I'll add it to the master copy of the code and will include it with the next Smoldyn version.

Table 3.6.2: statements about the command interpreter

`output_root` *str*	root of path for text output
`output_files` str_1 str_2 … str_n	file names for text output
`output_precision` *int*	precision for numerical output
`append_files` str_1 str_2 … str_n	file names for text output
`output_file_number` *int*	starting suffix number for file name
`cmd` b,a,e *string*	command run times and strings
`cmd` @ *time string*	
`cmd` n *int string*	
`cmd` i *on off dt string*	
`cmd` x *on off dt xt string*	

3.7 Surfaces

Surface basics

A large fraction of biochemistry does not happen in free solution, but at or across cellular membranes. To model these interactions, Smoldyn supports surfaces within the simulation volume. Typically, one Smoldyn surface is used to model each type of membrane. For example, a bacterium might be modeled with one surface for the inner membrane and another for the outer membrane, while a eukaryotic cell would use separate surfaces for the plasma membrane, the nuclear membrane, and for each type of organelle. Smoldyn supports disjoint surfaces as well, such as for a collection of vesicles.

Each Smoldyn surface comprises many panels. These panels have simple geometries: for three-dimensional systems, a panel may be a rectangle, triangle, sphere, cylinder, hemisphere, or a disk. For one- and two-dimensional systems, lower dimensional analogs of these panel shapes can be used. There are two main reasons that Smoldyn supports this variety of primitive shapes rather than just the triangle meshes that are more common. First, these are much easier to use for simple models. For example, it is much easier to specify a simple spherical nucleus for a cell than it is to build an approximate sphere out of 20 or more triangles. Second, it is faster to simulate molecular collisions with one sphere or other simple curved objects than with a lot of triangles. In general, more geometric primitives are better. (Although, from the Smoldyn programmer's point of view, each one also requires a significant amount of math before it can be supported by Smoldyn).

Each surface includes a set of rules that dictate how molecules interact with it. This includes molecules that diffuse into it from solution, as well as molecules that are bound to the surface. All panels on a single surface interact with molecules in the same ways. Molecules that are bound to a surface are designed to represent membrane-bound proteins and trans-membrane proteins. For example, they can model signaling receptors or ion channels.

Defining surfaces

Surfaces are typically entered with one or more blocks of statements that start with `start_surface` and end with `end_surface`. Between these, only surface statements are recognized. A single surface may be broken up into multiple blocks of statements, and each block may describe multiple surfaces. The surface name may be given after the `start_surface` statement, or it can be given afterwards with the `name` statement; this specifies which surface is being defined, and starts a new one if required.

As was mentioned before, Smoldyn surfaces do not work well in conjunction with the system boundaries that were defined with the `boundaries`, `low_wall`, or `high_wall` statements. If a configuration file includes any surface statement, even if no surfaces are actually defined, then all wall-type boundaries automatically behave as though they are transparent. To keep molecules within the system, an outermost surface needs to be defined. It may be a set of rectangular panels that are coincident with the system walls, a sphere that encloses the system, or something else. Molecules could also be allowed to

escape the system although that is usually undesirable and can slow the simulation down (see below for the `unbounded_emitter` statement, which provides an efficient alternative to escaping molecules).

The `action` or `rate` statements set the rules that molecules follow when they interact with a surface. For molecules in solution that collide with one of the surface faces, which are called front and back, there are three basic actions: reflection off of the surface, transmission through the surface, or absorption by the surface. It is also possible for a surface to be a "jumping" surface, such that if a molecule collides with it in one place, the molecule will be magically transported to a pre-defined destination. This is described below, as is another type of special surface called a "porting" surface. Yet another action option is "multiple", meaning that there any of several outcomes are possible and that there are specific rates for each. These rates are set with the `rate` statement (if `rate` is entered, the only possible action is "multiple", so the `action` statement may be omitted). For example, a membrane might be somewhat permeable to a molecular species, in which case one would set some rate for transmission; molecules that are not transmitted are reflected. Using the `rate` statement, it is also possible to cause a molecule to change species when it interacts with a surface. This is designed for molecules that behave sufficiently differently in different regions of space that it is most convenient to treat them with two different species; a typical use is for spatially-dependent diffusion coefficients.

The `action` and `rate` statements also apply to collisions of surface-bound molecules with other surfaces. This can arise when molecules diffuse along surfaces and two surfaces cross each other. For example, one way to create a lipid raft is to create a single surface for a cell membrane and then a short cylinder that intersects the membrane, creating an inner circular region and an other region (a Gaussian pillbox). Then, surface-bound molecules change species names when they cross the cylinder. An exception to the normal behavior arises when a surface-bound molecule collides with a panel that has been declared to be a neighbor of the molecule's panel. In this case, there are two options, which are selected with the `neighbor_action` statement. The default behavior is that the molecule simply ignores the panel and diffuses through it. Alternatively, the molecule can be allowed to hop onto the new panel, with a 50% probability of doing so. This latter possibility is helpful for allowing diffusion on a surface where the panels don't necessarily meet at their edges.

Sometimes, one wants a modeled system to be unbounded, such as for the simulation of pheromones that diffuse freely between cells, but that can also diffuse away towards infinity. While Smoldyn can simulate such unbounded systems with unbounded space, this can be very computationally intensive because it tracks every molecule, no matter how far it is from the region of interest. A better solution is to define a surface that surrounds the portion of the system that is of interest, where these surface panels absorb molecules at a rate that causes the system to behave as though it were unbounded. Smoldyn calculates this absorption rate automatically, from information that the user specifies with the `unbounded_emitter` statement. This statement declares the positions and the production rates for each emission source within the simulation volume. The new absorption coefficients completely replace any other actions that might be defined for interactions between this surface and molecular species.

Defining surface panels

Individual surface panels are defined with one `panel` statement for each individual panel. These statements are used to specify panel locations, dimensions, orientations, and, sometimes, drawing information. Each panel also has a name, for which the default is simply the panel shape followed a number, although it is also possible for the name to be defined by the user at the end of the `panel` statement. These names are used for jumping surfaces and diffusion of surface-bound molecules. For a surface to work in a consistent manner, it is worth making sure that all panel front sides face the same way. The drawing information, such as the numbers of slices and stacks for a sphere, is only used for graphical rendering. As far as the simulation is concerned, a sphere, regardless of how it is drawn, is always a mathematically perfect sphere.

In general, panels should not be defined that are coincident with each other because this can lead to unreliable behavior. The rule is that if multiple panels are *exactly* coincident, whether they are members of the same surface or different ones, then only the one that is defined last in the configuration file is in effect. For example, one could define a washer-shaped surface using a large disk that reflects all molecules and a small disk, which has the same center and is parallel to the large disk, that transmits all molecules. However, computer round-off error often makes exact coincidence impossible; at best, it is most likely to work if the panels are parallel to the system axes or if they share the same center point. If two panels are very nearly but not exactly coincident (separations between 0 and 10^{-12} distance units), Smoldyn treats them as though they are reflective, which it has to do in order to prevent unintentional leaks where panels cross each other. Graphical rendering of coincident panels is unpredictable but rarely good.

Several configuration files were written to test the surface actions with all dimensions and all panel shapes. They are in the examples/S7_surfaces directory and are called reflect#.txt, transmit#.txt, and absorb#.txt, where the '#' is 1, 2, or 3 for the system dimensionality. Additionally, the three surf#.txt files show the basic actions in single files. Following is an excerpt from reflect3.txt, which shows how a surface and its panels can be defined:

```
start_surface surf
action all both reflect
color both purple 0.5
thickness 2
polygon front face
polygon back edge
panel rect +0 40 40 40 20 20
panel rect -0 60 40 40 20 20
panel rect +1 40 40 40 20 20
panel rect -1 40 60 40 20 20
panel rect +2 40 40 40 20 20
panel rect -2 40 40 60 20 20
panel tri 60 15 70 80 15 70 70 15 86     # 1 2 3
panel tri 60 15 70 70 15 86 70 31 77     # 1 3 4
panel tri 70 15 86 80 15 70 70 31 77     # 3 2 4
panel tri 80 15 70 60 15 70 70 31 77     # 2 1 4
panel sph 20 20 20 8 20 20
panel cyl 20 75 20 80 75 80 5 20 20
```

```
panel cyl 20 30 70 20 50 70 4 20 20
panel hemi 20 75 20 5 1 0 1 20 20
panel hemi 80 75 80 5 -1 0 -1 20 20
panel disk 20 30 70 4 0 -1 0 20
panel disk 20 50 70 4 0 1 0 20
end_surface
```

Several statements pertain to the drawing of surfaces to the graphics window. The `color` statement specifies the color of the front and/or back of the surface with either color words or red, green, blue, and alpha (opacity) values. As mentioned above in the graphics section, OpenGL does not render well with alpha values between 0 and 1. `Thickness` defines the line width that should be used for drawing surface edges, or for surfaces in 2-dimensional systems. The `polygon` statement is used to set the drawing mode for showing just the panel edges, only panel vertices, or complete panel faces. It also allows filling of regions for surfaces in 2-dimensions.

Jumping surfaces

There are a few situations in which one might reasonably want to have molecules move discontinuously, leaping from one place to another. One is for periodic boundaries in which molecules that diffuse off of one side of the system immediately diffuse onto the other side, thus keeping the composition of the system constant while avoiding effects that can arise from edges. Another situation is for building complex surface structures from the Smoldyn panel primitives without resorting to triangulated meshes. For example, one might want to have two spherical cells whose cytoplasms are linked by a narrow cylindrical channel, making a dumbbell shape. This would be easy to design in Smoldyn, except that there is no way to cut holes in the spheres where the cylinder should be attached. The solution is to put small disk-shaped "jumping" panels on each side of the spot where the hole is wanted so that molecules can be transported across the barrier (see examples/S7_surfaces/dumbbell.txt).

To define a jumping surface, the action for each molecule that is to be jumped (usually set to all molecules, although fewer is permissible too), for the active face of the surface, is set to "jump." Next, the active face of each panel needs to be assigned a destination panel and face using the `jump` statement. The source and destination panels are required to be the same shape and to be parallel to each other although, for certain shapes, they may differ in size.

Jumping surfaces are demonstrated with the files jump1.txt, jump2.txt, and jump3.txt, all in the S7_surfaces directory.

Surface-bound molecules used to jump when they diffused onto panels that had surface-bound jump actions. However, this feature was removed in version 2.37 because it was complicated and there were better ways of accomplishing the same result.

Membrane-bound molecules

In Smoldyn, molecules can be in free solution or bound to surfaces. The bound ones can be attached on the front of the surface or on the back, called the "front" and "back" states, or they can be transmembrane molecules in either an "up" orientation or a

"down" orientation. The precise meanings of these states are decided by the user. As an example, if a receptor is oriented such that the ligand binding site is on the outside of the cell, as usual, it could be called "up," whereas if it were in the membrane in a reversed orientation, it could be called "down." In all, there are five states that molecules can be in: "solution," "front," "back," "up," or "down," of which the last four are the surface-bound states. In practice, all four surface-bound states are essentially equivalent. A molecule in any of these states is allowed to interact with solution-phase molecules that are on either side of the surface, and it can desorb to either side of the surface. The only real difference between these states is that Smoldyn ensures that molecules in the "front" state have coordinates that are slightly on the front side of the surface and those in the "back" state have coordinates that are slightly on the back side of the surface. Smoldyn does not fix the coordinates to be on any particular side for molecules in the "up" or "down" states, which makes these states simulate very slightly faster.

Additionally, it is sometimes necessary to specify the position of a solution-state molecule relative to a surface. For this, the pseudo-states "fsoln" (which is identical to "solution") and "bsoln" specify that it is solution state and on the front or back of the relevant surface.

The `surface_mol` statement, which was mentioned in the section on molecules, is used to specify that there are molecules bound to a surface at the start of a simulation. The statement is quite versatile, allowing one to specify that molecules are scattered randomly over an entire surface, over specific panel shapes, over specific panels, or even over all surfaces. Also, of course, it is possible to specify exact molecule locations.

The `rate` statement, mentioned before in the context of partially permeable surfaces, is also used for transition rates for surface-bound molecules. It can be used for specifying the rate at which a solution-state molecular species is adsorbed onto a surface. It can also be used for the release rate, from surface to solution. In this situation, the release side of the surface is identified by giving the destination state as either "fsoln" or "bsoln", for the front and back, respectively. `Rate` is also used for transition rates between the different surface-bound states, such as from "front" to "back."

Surface-bound molecules diffuse within the plane of the surface according to the diffusion coefficient that was entered with the `difc` statement for the respective molecule state. To allow molecules to diffuse between neighboring surface panels, whether they are part of the same surface or different surfaces, these neighbors have to be declared with the `neighbors` statement. Diffusion on surfaces is reasonably quantitatively accurate, which is best understood with an explanation of the algorithm (most of which was new in version 2.37). Considering a three-dimensional system, a surface-bound molecule is initially diffused in all three dimensions. It is then moved back to the local plane of the panel that it is bound to. If this puts the molecule within the area of its panel, then the diffusion step is done and no further actions are taken. This approach is exact for flat panels and reasonably good for curved panels (and becomes exact in the limit of short time steps). If the new position is not within the area of the molecule's panel, Smoldyn determines where the line of the molecule's trajectory exits the current panel. Smoldyn then determines if there are other panels at this point (it actually checks for panels within an extremely small distance called *neighdist* from this position, which is just large enough to prevent problems from computer round-off error). If so, it chooses one of these panels at random and rotates the molecule's trajectory that extends beyond

the original panel into the plane of the new panel, thus preserving the length of the trajectory. If the end point of the new trajectory is within the new panel, then the diffusion step is done. If not, Smoldyn repeats the procedure until the trajectory is used up. Returning to a prior condition, if the molecule's trajectory leaves the molecule's current panel but there is no neighbor near the exiting point, then the molecule does not continue onto a neighbor. Instead, it reflects off of the panel edge, so that the trajectory continues on the original panel. This procedure should be exact for flat panels and extremely good for curved panels.

Note that molecules only transition from one panel to another when they diffuse off the edge of the initial panel. Thus, for example, a molecule can never diffuse off an edge of a sphere, with the result that molecules cannot diffuse from one sphere to another, even if these spheres intersect. If diffusion between panels is desired in these cases, then use the `neighbor_action` statement, as described above. However, be forewarned that diffusion between neighboring panels can interact badly with the `neighbor_action` hopping, which is why this hopping is turned off as a default. For example, suppose several 2D panels (which are lines) meet at a single point. A molecule diffusing along one of the panels correctly transitions to a new randomly chosen panel when it gets to that point. However, if `neighbor_action` is set to hopping, then the trajectory during this transition might be discovered to cross yet another one of the panels in the process, so the molecule would then get moved onto this new panel. The probability of this outcome is biased by the precise panel positions and by round-off errors, with the result that the molecule position statistics would be incorrect.

Files that demonstrate surface-bound molecules are: S7_surfaces/stick2.txt and cellmesh.txt (which reads cellmeshfile.txt). Surface diffusion is demonstrated with the files in S7_surfaces/surfacediffuse/.

Smoldyn bugs

As far as I know, there are no bugs currently in Smoldyn that cause surfaces to behave other than requested. However, leaking surfaces have been a recurring problem with Smoldyn. In this problem, which can be caused by any of a vast number of small mistakes in the source code, molecules that shouldn't go through a surface are found to have done so. Some commands that were written to test for it are: `warnescapee` and `killmoloutsidesystem`. If you suspect that Smoldyn isn't working right, or if you just want to verify that it is working right (a good idea if you don't use graphical output), then it might be worth running these or other commands. The former one has to be run at every time step to be useful. The latter one has no output directly, but will identify problems if it is bracketed by `molcount` commands. The command `killmolinsphere` can be used in a similar manner.

If you have a configuration file that shows molecules leaking through surfaces incorrectly, please send it to me (steven.s.andrews@gmail.com), so I can track down the bugs. Note that simulations will produce repeatable results, which is essential for debugging, if the statement `random_seed` is used to fix the random number seed.

Table 3.7.1: statements about surfaces

`max_surface` *int* (optional) maximum number of surfaces
`start_surface` *name* start of a surface block
`name` *name* optional statement for the surface name
`action` *species(state) face action* action for when a molecule contacts surface
`rate` *molec state1 state2 value* [*new_spec*] transition rate
`neighbor_action` *action*
`rate_internal` *molec state1 state2 value* [*new_spec*]
`color` *face color* [*alpha*]
    `color` *face red green blue* [*alpha*]
`thickness` *float*
`polygon` *face drawmode*
`shininess` *face value*
`max_panels` *char int* (optional)
`panel` *char float … float*
    `panel` *char float … float name*
`jump` *name face -> name2 face2*
    `jump` *name face <-> name2 face2*
`neighbors` *panel neigh1 neigh2 …*
`unbounded_diffusion` *face species amount* $pos_0\ pos_1 \ldots pos_{dim\text{-}1}$
`end_surface`

Rates of surface interactions

For an interaction to occur between a solution-state molecule and a surface, the molecule has to (1) contact the surface and (2) interact based on some probability. There are subtleties both in the determination of contacts and in the calculation of these probabilities.

Starting with the contacts, a molecule clearly contacted a surface during the preceding time step if it ended up across the surface from where it began, which I'll call a direct collision. It is also possible for a molecule to start and end on the same side of a surface, but to have contacted the surface at some point during the time step, labeled here as an indirect collision. The probability of an indirect collision occurring is (Andrews and Bray, *Phys. Biol.* 2004)

$$\exp\left[-\frac{l_1 l_2}{D\Delta t}\right]$$

Here, l_1 and l_2 are the perpendicular distances to the surface before and after the time step, D is the diffusion coefficient, and Δt is the time step. These indirect collisions are implemented in Smoldyn for simulating absorption of molecules to the bounding walls of the system (the boundaries).

However, for interactions between diffusing molecules and all surfaces, Smoldyn only accounts for direct collisions, thus ignoring the indirect collisions. This decreases the accuracy of Smoldyn slightly but is done because indirect collisions were found to be

difficult to code, computationally demanding, and made essentially no difference to results.

The probability of interaction given that a collision has occurred is difficult to calculate. While it is presented in a recent paper by Erban and Chapman (*Phys. Biol.* 4:16-28, 2007) for adsorption interactions, their equation turns out to only be accurate in the limit of short time steps. Thus, I found the necessary relationships between the adsorption, desorption, or transmission coefficients and the corresponding adsorption, desorption, and transmission probabilities. They are implemented in the SurfaceParam.c source code file of Smoldyn and have been thoroughly tested. I plan to write these algorithms up and submit them for publication during the next few months.

The adsorption coefficient, κ, has units of length/time. The product κc, where c is a concentration (units of length^{-3}), is the adsorption rate in molecules adsorbed per unit of time, per unit of surface area. If the surface is in equilibrium with the solution, where there is a sticking coefficient of κ, and an unsticking rate of k, then the equilibrium surface density of molecules is

$$C_{surface} = \frac{\kappa}{k} C_{solution}$$

Surface sticking rates were tested with the example file stickrate.txt. Here, a collection of molecules diffuses freely in solution, but sticks with rate 0.5 on one side. This situation can be solved analytically as well from equations in Crank, allowing for a good comparison. Comparison between simulation and theory are shown in the figure below.

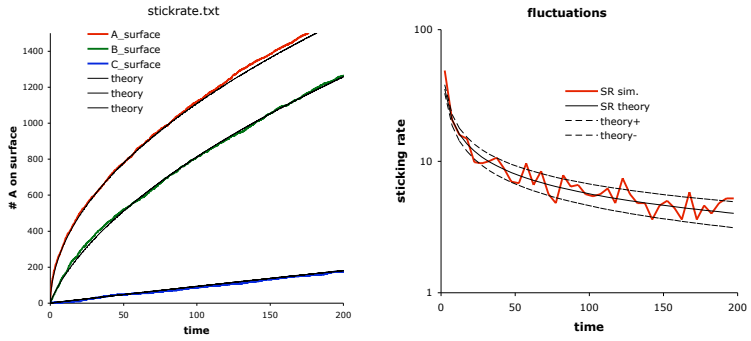

Figure 3.7.1: Sticking rate. Results from example stickrate.txt, shown in red, are compared with the analytic solution for the sticking rate. The left panel shows the total number of molecules stuck to the surface. The right panel shows the average sticking rate with a 5 time unit averaging window, with comparisons to the expectation sticking rate shown with a solid line and the 1 standard deviation range shown with dashed lines.

Simulating effective unbounded diffusion

The example files in `S7_surfaces/unbounded_diffusion` illustrate and verify the use of a partially absorbing bounding surface to simulate effective unbounded diffusion.

These files use the Smoldyn file `sphere.txt`, which describes a sphere; I created it by using Mathematica to define a sphere, triangulate it, and save it as a "wrl" (Virtual Reality Modeling Language) file. Then, I used the wrl2smol utility program to convert it to the Smoldyn-readable file `sphere.txt`. Other Smoldyn configuration files specify either one or multiple emitters within this sphere and then save concentration line profiles as functions of time. The theoretical concentration distributions for these situations is expressed with a slight extension of eq. 3.5b from Crank, which leads to

$$C(\mathbf{r}) = \sum_i \frac{q_i}{4\pi D|\mathbf{r}-\mathbf{r}_i|} \operatorname{erfc} \frac{|\mathbf{r}-\mathbf{r}_i|}{2\sqrt{Dt}}$$

Here, $C(\mathbf{r})$ is the concentration at position $\mathbf{r}$, q_i is the emission rate of source i, D is the diffusion coefficient, $\mathbf{r}_i$ is the location of source i, and t is the time since the sources started emitting. At steady-state, this concentration equation simplifies to

$$C(\mathbf{r}) = \sum_i \frac{q_i}{4\pi D|\mathbf{r}-\mathbf{r}_i|}$$

The figure below shows results from the `emitter1.txt` Smoldyn simulation, in which an emitter at location $\mathbf{r}_1$ = (-4.5,0,0) microns emits q_1 = 500 molecules per second, these molecules have a diffusion coefficient of D = 3 μm^2/s, and the system is surrounded by a triangulated sphere that is centered at the origin and has radius 10 microns. Absorption to this sphere was set to make the molecules diffuse as though the system were unbounded. Close agreement between simulation and theory show that the algorithm works well.

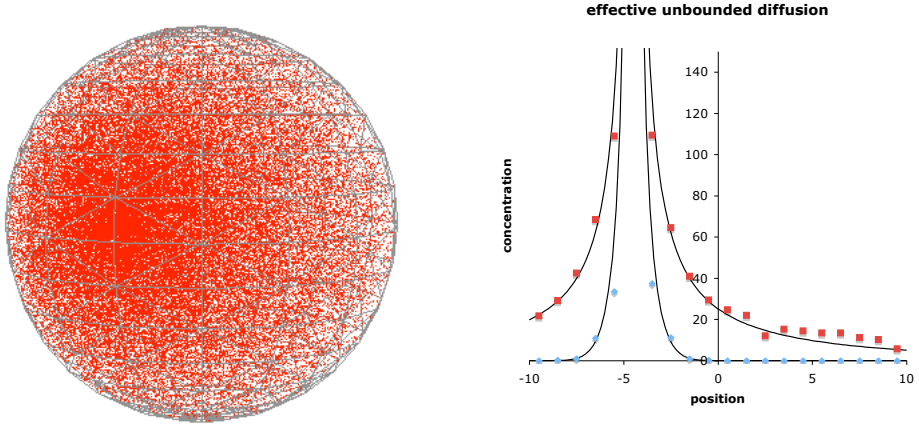

Figure 3.7.2: Effective unbounded diffusion. The left panel shows a snapshot from example emitter1.txt where it is seen that the emitter center is somewhat left of the sphere center and the sphere is triangulated. The right panel shows line profiles across the middle of the sphere, from (-10,0,0) to (10,0,0) at times t = 0.3 s (blue) and t = 100 s (red), with simulation data shown with points and theoretical results, from the equations above, in solid lines.

3.8 Reactions

Reaction basics

There are three types of chemical reactions in Smoldyn: zeroth order, first order, and second order, where the order is simply the number of reactants. Synonyms for the latter two are unimolecular and bimolecular reactions. In addition, Smoldyn simulates a couple of additional interaction types using reactions; these are conformational spread reactions and excluded volume reactions, both of which are described below.

With zeroth order reactions, there are no reactants at all. Instead, products appear spontaneously at random locations in the system volume (or within a compartment) at a roughly constant rate. This is completely unphysical because particles are being created from nothing. However, since Smoldyn explicitly ignores many chemical species, the assumption here is that some unmodeled chemicals are being converted into the zeroth order product. Thus, it is assumed that there is a legitimate underlying chemical reaction that produces the products that are seen, but it just isn't part of the model. (At least, this is the typical use of zeroth order reactions; using them to model the magical production of matter is fine too.)

First order reactions involve the conversion of one molecular species into another. This includes spontaneous conformational changes of proteins and chemical rearrangements of small molecules. Also, many reactions are pseudo-first order, meaning that one of two reactants has a sufficiently constant concentration and distribution that it can be left out of the model and its effect is lumped into the rate constant of a first order reaction. Protein phosphorylation by ATP is a good example of this. In Smoldyn, reactants of first order reactions have a certain probability of converting to products at each time step.

Second order reactions occur when two reactants collide and react (conformational spread reactions are an exception, as described below). In Smoldyn, a reaction radius is defined for each pair of molecular species. For those that do not react with each other, the reaction radius is 0. For those that can react, the reaction radius is some small distance on the order of the molecular radii, with values that increase monotonically with the standard mass action reaction rate. To simulate each time step, molecules are first diffused and then all reactant pairs that are closer than their reaction radii are reacted. Thus, the stochasticity in simulated bimolecular reactions arises solely from diffusion and not from the reaction step of the algorithm. This is slightly less accurate than alternative methods in which there is a non-one probability of reaction upon collision but is chosen because it is nearly as accurate and is faster to simulate.

If a reaction has multiple products, they are usually all added to the system at the same point. They can also be separated from each other by a small amount, called the unbinding radius if there are two products, which reduces the likelihood of their immediate recombination in a new reaction. This recombination is called a geminate recombination.

It is possible to specify that a reaction should only occur within a spatial compartment (defined below), or if one of the reactants is bound to a specified surface. For example, it is possible to declare that a zeroth order reaction should only produce product within a specific compartment, or that a first order reaction is only active when

the reactant is within the specified compartment. In many cases, these rules are unphysical, although they can be very useful for treating interactions with spatially localized unmodeled chemical species.

Conformational spread reactions are only intended to be used with stationary reactants and are only permitted in reactions with two reactants and two products. A conformational spread reaction is possible if the reactants are closer together than the conformational spread radius, which is analogous to the binding radius of normal second order reactions (although its value is constant, regardless of the time step). For a conformational spread reaction, the reaction rate has units of inverse time, as it is for a first order reaction. If a reaction occurs, the first entered reactant is replaced by the first product, and the second reactant with the second product.

Excluded volume reactions use the reaction concept to simulate excluded volume interactions. Here, the typical reaction is of the form $A + B \rightarrow A + B$, but with the constraints that the "binding radius" is roughly the sum of the physical molecular radii and the "unbinding radius" is a slightly larger value. In these reactions with the "bounce" product placement type, the reaction products are placed on the same axis as the reactants, with the result that the molecules appear to bounce off of each other.

Each molecule has a serial number that can be used to uniquely identify it. In most reactions, the reactants are simply removed from the system and the reaction products are new molecules with new serial numbers. However, this is not the case for conformational spread and excluded volume reactions because the reactants and products are conceptually the same molecules, so these products have the same serial numbers as the reactants. It can also be helpful to maintain serial numbers in other situations, such as for single molecule tracking. In these situations, use the `reaction_serialnum` statement to define rules for the product serial number assignments.

Defining reactions

To define a reaction, simply enter the statement `reaction`, followed by the reaction name, the reaction, and the rate constant. Here are some examples:

```
reaction r1 A + B -> C 10
reaction bind receptor(up) + ligand(fsoln) -> complex(up) 1
reaction ingest complex(up) -> receptor(up) + ligand(bsoln) 5
reaction tca 0 -> ATP 100
reaction decay fluorophore(all) -> 0 0.01
```

For molecule states that are not specified, as in the first example above, it is assumed that the reaction only applies to molecules that are in solution. Reactions that only occur in specified compartments are entered in the same way, but with the `reaction_cmpt` statement. Versions of Smoldyn prior to 1.82 allowed reactions to be entered in definition blocks; this is still permitted for backward compatibility, but is discouraged because this format is not being maintained and may be eliminated in future versions.

For most applications, the `reaction` statement is sufficient for entering the reaction rate. However, other methods are possible as well. It is possible to leave the rate constant off of the reaction line and enter it separately with the statement `reaction_rate`. The reaction rate is the macroscopic reaction rate, which is converted into parameters that

Smoldyn can use for the simulation. For zeroth order reactions, the reaction rate is converted to the average number of molecules that should be added to the entire simulation volume at each time step. To enter this internal value directly, use the statement `reaction_production`. For first order reactions, the reaction rate is converted to the probability that a reactant molecule will react during one time step. This can be entered directly with the statement `reaction_probability`. For second order reactions, the reaction rate is converted into a reaction binding radius, which can be entered directly with `binding_radius`.

In most cases, the states that are listed with the reaction statement are all that are desired. For a reaction to work with additional states, it is possible to simply list another reaction that has those states. Alternatively, it is also possible to enable or disable the reaction for specific molecule states, or state combinations for bimolecular reactions, with the `reaction_permit` and `reaction_forbid` statements.

If a reaction has multiple products, they are usually placed at the location where the reaction was determined to have occurred. However, offsets from the reaction location are possible as well, which are necessary for reversible reactions so as to avoid certain geminate recombinations. Offsets can be entered directly or can be calculated by Smoldyn in many different ways. All of them are entered with the `product_placement` statement.

Conformational spread reactions are a special type of bimolecular reactions. For these, there is a domain of interaction, which is entered with the statement `confspread_radius`; this also specifies that the reaction uses conformational spread. Reaction rate constants for conformational spread reactions have units of inverse time, like a first order reaction rate constant. They indicate the rate at which a reaction occurs, for reactants that are continuously closer to each other than the conformational spread radius. As with first order reactions, this rate value is converted to a reaction probability at each time step, and can be entered directly with the `reaction_probability` statement. The two products of conformational spread reactions are placed at the exact same locations as the two reactants, using the same ordering of reactants and products as they are listed with the `reaction` statement.

Table 3.8.1: statements about reactions

`reaction` *rname reactant*$_1$ + *reactant*$_2$ -> *product*$_1$ + *product*$_2$ *rate*
`reaction_cmpt` *cname rname reactant*$_1$ + *reactant*$_2$ -> *product*$_1$ + *product*$_2$ *rate*
`reaction_surface` *sname rname reactant*$_1$ + *reactant*$_2$ -> *product*$_1$ + *product*$_2$ *rate*
`reaction_rate` *rname rate*
`confspread_radius` *rname rad*
`binding_radius` *rname rad*
`reaction_probability` *rname prob*
`reaction_production` *rname value*
`reaction_permit` *rname state*
    `reaction_permit` *rname state*$_1$ *state*$_2$
`reaction_forbid` *rname state*
    `reaction_forbid` *rname state*$_1$ *state*$_2$

`reaction_serialnum` *rname rules_list*
`product_placement` *rname type parameters*

Although now discouraged, the block format for entering reactions is similar. The block starts with the statement "`start_reaction`" and ends with "`end_reaction`", between which only instructions that are relevant to reactions are allowed. The first statement within a reaction block is `order` to define the reaction order of this block. The `max_rxn` statement used to be required next, but is no longer functional as of version 1.82. Basic reactions are entered with a `reactant` statement, a `rate` statement, and a `product` statement. It is also possible to enter the internal value that Smoldyn uses with `rate_internal`. It is possible to turn states on or off with the `permit` statement. If there are multiple products, and if these products can react with each other (most often a reversible reaction), then Smoldyn may need some information about the product unbinding radii, which is entered with the `product_param` statement. It is discussed at length below.

Conformational spread reactions are slightly different. Enter the conformational spread radius with the `confspread_radius` statement and the reaction rate (which is analogous to a first order rate) with `rate`. This rate value is converted to a reaction probability at each time step. To enter the latter value directly, do so with the `probability` statement. The `rate_internal` statement is ignored.

Zeroth order reactions

Zeroth order reactions have no reactants and yet produce products at a rate that is constant except for stochastic fluctuations. They can be used to simulate the production of molecules that are of interest from sub-systems that are not of interest and thus are not explicitly part of the model. As mentioned above, zeroth order reactions have not proven to be particularly useful.

The zeroth order reaction $0 \rightarrow \mathrm{A}$ proceeds according to the mass action rate equation

$$\frac{d[\mathrm{A}]}{dt} = k$$

k is the reaction rate constant. Solving for the number of A molecules in volume V as a function of time yields the deterministic solution

$$n(t) = n(0) + kt$$

$n(0)$ and $n(t)$ are the initial and time dependent numbers of A molecules. There are also fluctuations due to the stochastic nature of chemical processes. Smoldyn assumes that each molecule created in a zeroth order reaction is created independently of each other, which allows Poisson statistics to be used. As an example of a limitation, this is not a perfect description of biochemical protein production because that involves sequential

stochastic DNA transcription followed by many relatively rapid mRNA translations, thus leading to stochastic bursts of protein production.

Zeroth order reactions were tested with the file zeroreact.txt. The reaction portion of the configuration file is

```
reaction slow 0 -> red 0.001
reaction med  0 -> green 0.01
reaction fast 0 -> blue 0.1
```

As seen in the figure below, simulation results conform closely to corresponding theoretical results, using a wide range of reaction rates. As expected, stochastic deviations from the deterministic theoretical predictions are seen.

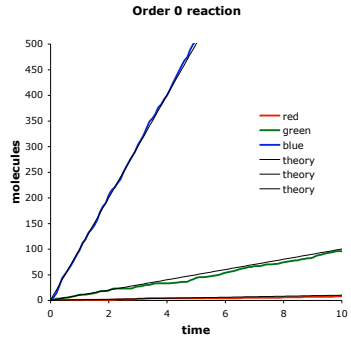

Figure 3.8.1: Zeroth order reaction molecule production with data simulated from the example file S8_reactions/zeroreact.txt. Shown are the numbers of molecules produced as a function of time with three different production rates along with the deterministic theory for how many molecules would be expected.

Unimolecular reactions

Order 1 reactions follow the general reaction equation $A \rightarrow B$. The mass action kinetics for the loss of reactant are described with the differential equation

$$\frac{d[A]}{dt} = -k[A]$$

k is the first order reaction rate. This is solved to yield the deterministic solution for the number of A molecules as a function of time,

$$n(t) = n(0)e^{-kt}$$

$n(0)$ is the number of A molecules at time 0 and $n(t)$ is the number at time t.

The example file S8_reactions/unireact1.txt was used to check unimolecular reaction rates using a wide range of reaction rates. The reaction portion of the configuration file is

```
reaction slow red -> 0 0.1
reaction med green -> 0 1
reaction fast blue -> 0 10
```

As seen in the figure below there is good agreement between simulation and theory. As always, stochastic fluctuations are apparent, which is particularly true when there are few molecules.

First order reactions in which a reactant can react through multiple possible pathways requires slightly more complicated calculations for the reaction probabilities. However, the mass action differential equation, shown above, is unchanged. This situation was tested with the configuration file unireactn.txt. The reaction portion of the configuration file is

```
reaction r1 A -> A + B 0.1
reaction r2 A -> A + C 0.05
reaction r3 A -> A + D 0.01
```

The system is started with only A molecules, so the theoretical number of A molecules as a function of time is

$$n_{\mathrm{A}}(t) = n_{\mathrm{A}}(0)e^{-(k_{\mathrm{B}}+k_{\mathrm{C}}+k_{\mathrm{D}})t}$$

The number of B molecules as a function of time is

$$n_B(t) = n_A(0)\frac{k_B}{k_B + k_C + k_D}\left[1 - e^{-(k_B+k_C+k_D)t}\right]$$

Analogous equations hold for C and D. Simulation results closely matched these theoretical equations, as shown in the figure below.

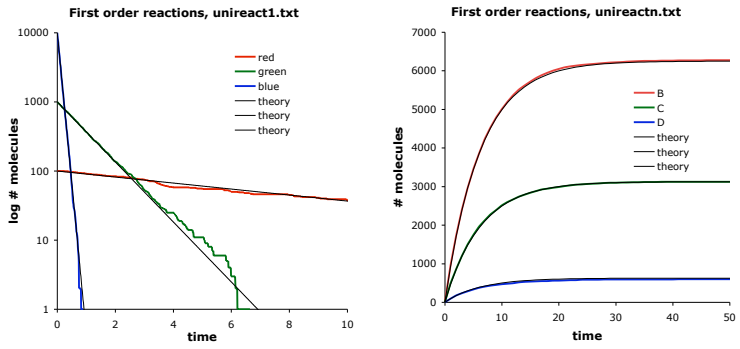

Figure 3.8.2: Unimolecular reactions. The panel on the left shows results from the configuration file unireact1.txt. First order reactions occur at rates that are in good

agreement with theory over a wide range of rate values. The panel on the right shows results from the file unireactn.txt. Again, there is good agreement with theory.

Bimolecular reactions

Bimolecular reactions have the generic reaction equation A + B → C, for which the mass action kinetics are described by the deterministic differential equations

$$\frac{d[\mathrm{A}]}{dt} = \frac{d[\mathrm{B}]}{dt} = -\frac{d[\mathrm{C}]}{dt} = -k[\mathrm{A}][\mathrm{B}]$$

The reaction rate constant, k, is only actually constant if: (*i*) the reaction kinetics are purely activation-limited, or (*ii*) the reaction has proceeded long enough that a steady-state reactant distribution has formed.

This equation is not quite as trivial to solve as prior ones were. With the condition that there are the same numbers of A and B molecules initially, the solution for the number of A molecules (or B molecules) as a function of time is

$$n(t) = \left(\frac{1}{n(0)} + \frac{kt}{V} \right)^{-1}$$

As before, $n(0)$ is the initial number of A or B molecules, $n(t)$ is the number of A or B molecules as a function of time, k is the reaction rate constant and V is the volume of the system. This was tested with three different reaction rates with the configuration file reactAB.txt, for which the reaction portion of the file is

```
reaction slow As + Bs -> Cs 1
reaction med  Am + Bm -> Cm 10
reaction fast Af + Bf -> Cf 100
```

The Smoldyn diagnostics output shows how these different reaction rates are converted into simulation parameters. They are converted into binding radii, which is small for the slow reaction and large for the fast reaction. Because the reaction kinetics depend on the ratio of the reactant rms steps lengths to the binding radii, the slow one has relatively long steps compared to the binding radius and thus behaves as though it is activation-limited. In contrast, the fast reaction has short rms step lengths compared to the binding radius and so behaves as though it is diffusion-limited. Shortening the simulation time step would make all of these more diffusion-limited.

Activation-limited reactions follow the mass action kinetics shown in the equations for all times. Thus, the slow and medium reaction rate simulations agree well with the mass-action theory, as shown in the figure, below. In contrast, the diffusion-limited simulation does not agree with the mass-action theory. This is because the simulation starts with molecules randomly distributed whereas the analytical result assumes a steady-state distribution. However, after enough time has passed for a steady state

reactant distribution to be formed, it is shown that the simulated results agree well with the analytical results (orange line in the figure).

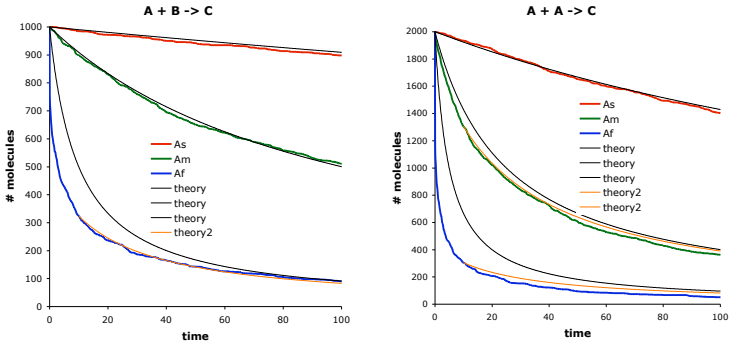

Figure 3.8.3: Bimolecular reactions. The panel on the left shows reactant numbers for the reaction A + B → C for three different reaction rates and with equal initial numbers of A and B molecules. The panel on the right is similar but for the reaction 2A → C. Light black lines are solutions to the deterministic steady-state mass action rate equations. Deviations arise for the faster reactions (blue lines) because those start far from steady-state. Light orange lines are the steady-state theory, starting with time 10 rather than 0, so as to start at times when reactants are closer to steady-state distributions.

Although there are no conceptual or simulation algorithm differences for bimolecular reactions in which two reactants are the same, there are a few quantitative differences. Consider a situation with 1000 A molecules and 1000 B molecules. Despite the fact that each A molecule has about 1000 potential collision partners, whether the reactants are A + A or A + B, there are twice as many A-B collisions as A-A collisions. This is because each A-A pair can be counted in either of two ways, but is still only a single possible collision. To achieve the same reaction rate for A + A reactants as for A + B, despite the fact that there are fewer collisions, Smoldyn uses a larger binding radius for the former.

The analytical solution for the number of A molecules as a function of time is also slightly different from before,

$$n(t) = \left(\frac{1}{n(0)} + \frac{2kt}{V} \right)^{-1}$$

The reaction description portion of the configuration file S8_reactions/bireactAA.txt is

```
reaction slow As + As -> C 1
reaction med  Am + Am -> C 10
reaction fast Af + Af -> C 50
```

Results are similar to those seen before. Simulation results agreed well with the analytical equations if the reaction is activation-limited or once the reactant distributions have reached steady-state, but agreement is not good for diffusion-limited reactions away from steady-state. It should be emphasized that these discrepancies are not errors by Smoldyn, but are quite the opposite: they are approximations made in the steady-state equations which people are used to making but which are nevertheless incorrect, which are being compared to accurate simulations by Smoldyn.

Diffusion-limited reactions can be simulated well by Smoldyn. The example file bireactABB.txt again simulates the reaction A + B → C, but now with a lot more B molecules than A ones, and with a time step that is sufficiently short that the reaction simulates as though it is diffusion-limited. As is shown in the figure below, results conform closely to the Smoluchowski prediction for this reaction.

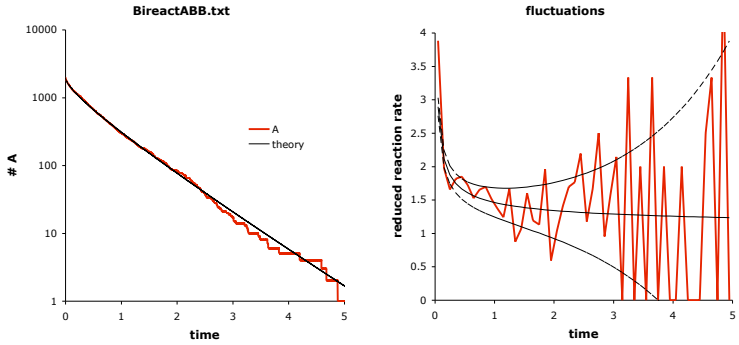

Figure 3.8.4: Diffusion-limited bimolecular reaction. This example uses the configuration file bireactABB.txt, which simulates the reaction that is described in Figure 7 of Andrews and Bray, 2004. The left panel shows the number of surviving A molecules as a function of time with comparison to the time-dependent Smoluchowski equation. The right panel shows the reaction rate per A molecule per time unit as a function of time along with the Smoluchowski prediction with the solid black line and predicted fluctuations with the dashed lines.

Reversible reactions

Reversible reactions, where at least one has multiple products, involve geminate recombination issues, as discussed below. The accuracy of reversible reaction rates using the default reverse parameter type and parameter was investigated with the configuration file S8_reactions/equil/equil.txt. Here, an equilibrium is set up for the reaction A + B ↔ C.

From standard chemistry, the equilibrium constant is related to the ratio of product to reactant concentrations and to the ratio of the forward to reverse rate constants,

$$K = \frac{n_C V}{n_A n_B} = \frac{k_f}{k_r}$$

V is the total system volume. The configuration file equil.txt starts with equal numbers of A and B molecules and no C molecules. Using the above equation and this starting point, the solution for the equilibrium number of A molecules is

$$n_{\mathrm{A}} = \frac{-V + \sqrt{V^2 + 4Kn_{\mathrm{A}}(0)V}}{2K}$$

$n_{\mathrm{A}}(0)$ is the initial number of A molecules. It was verified that the simulation result approached this value.

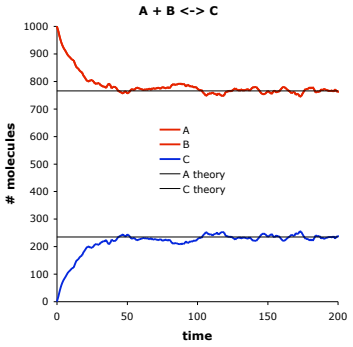

Figure 3.8.5: Equilibrium result from example file S8_reactions/equil/equil.txt.

Multi-step reactions

Many biochemical models include reactions that do not fall neatly into the 0th, 1st, or 2nd order reaction categories, but are instead complex reactions that include multiple elementary steps. Whereas these complex reactions can be well-defined for models that are either deterministic or non-spatial, they simply don't make sense when individual molecules are modeled. Thus, to include them in a Smoldyn model, one has to explicitly define each of the steps.

Taking the Michaelis-Menten reaction as an example, consider substrate S, enzyme E, and product P. The full reaction system is

$$\mathrm{E} + \mathrm{S} \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} \mathrm{ES} \xrightarrow{k_2} \mathrm{P}$$

All three of these reactions, along with the enzyme-substrate complex ES, need to be defined in a Smoldyn file. Of course, this means that you also need to give the three reaction rate constants k_1, k_{-1}, and k_2. Assume you know the Michaelis constant K_M and the maximum reaction velocity V_{max}. As can be found in any biochemistry textbook, these are connected to the underlying rate constants as

$$K_M = \frac{k_{-1} + k_2}{k_1} \qquad V_{\max} = k_2 [\mathrm{E}]_0$$

where $[\mathrm{E}]_0$ is the total enzyme concentration. These two equations are not sufficient to solve for the three rate constants, so let us define the unitless reaction efficiency ratio, r, as the fraction of ES that goes to P,

$$r = \frac{k_2}{k_{-1} + k_2}$$

This value can range between 0 and 1, where small values represent rapid equilibration between E, S, and ES, and high values represent rapid reaction of ES to P. Typical Michaelis-Menten analyses assume the former situation, so we might guess that r is 0.1. Solving these equations for the reaction rate constant yield:

$$k_1 = \frac{V_{\max}}{[\mathrm{E}]_0 K_M r} \qquad k_{-1} = \frac{V_{\max}(1-r)}{[\mathrm{E}]_0 r} \qquad k_2 = \frac{V_{\max}}{[\mathrm{E}]_0}$$

Other multi-step reactions can be broken down to elementary reactions in a similar manner. The need to include additional assumptions, as we did here with r, is typical when converting from a low-detail reaction rate equation to a high-detail reaction mechanism.

Reaction networks

The reaction types presented above can be combined to create essentially unlimited varieties of reaction networks. A particularly simple one is shown here as an example. It is the classic Lotka-Volterra reaction network, which was originally designed to explain observed oscillations in ecological predator-prey systems but is also analogous to many natural biochemical oscillators. The terminology used here borrows from the ecology application, although all numbers were chosen solely to make for an interesting simulation result. The complete file S8_reactions/lotvolt/lotvolt.txt is:

```
# Simulation file for Lotka-Voltera reaction

graphics opengl
graphic_iter 5

dim 3
names rabbit fox
max_mol 20000
molperbox 1

difc all 100
color rabbit 1 0 0
```

```
color fox 0 1 0
display_size rabbit 2
display_size fox 3

molecule_lists rlist flist
mol_list rabbit rlist
mol_list fox flist

time_start 0
time_stop 100
time_step 0.001

boundaries x -100 100 p
boundaries y -100 100 p
boundaries z -10 10 p

mol 1000 rabbit u u u
mol 1000 fox u u u

cmd b pause
#output_files lotvoltout.txt
#cmd i 0 5 0.01 molcount lotvoltout.txt

reaction r1 rabbit -> rabbit + rabbit 10
reaction r2 rabbit + fox -> fox + fox 8000
reaction r3 fox -> 0 10

end_file
```

This involves several statements that make the simulation run efficiently. Graphics are only displayed every 5 iterations, the simulation is set up with only 1 molecule per virtual box, and the rabbit and fox molecules are stored in separate molecule lists. Results from this file are shown in the figure below.

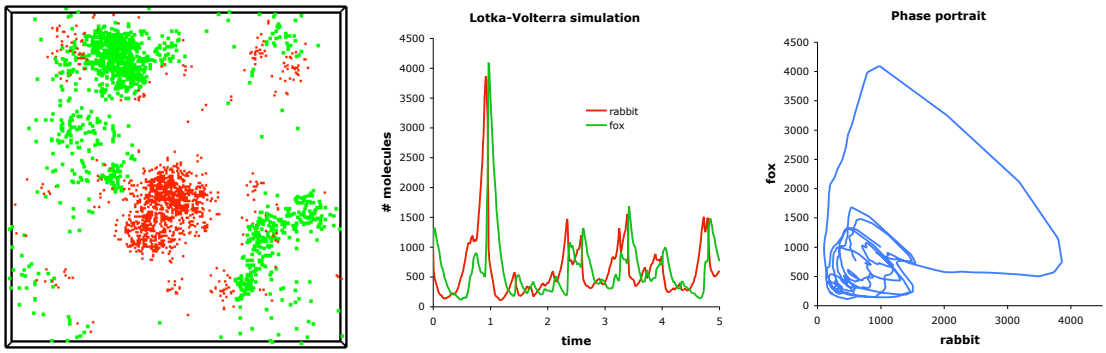

Figure 3.8.6: Results from Lotka-Volterra simulation. The first panel shows of snapshot of the simulation after it has run for long enough for the regular boom-and-bust pattern to develop. Red dots are "rabbits" and green dots are "foxes". The next panel shows the numbers of "rabbit" and "fox" molecules as a function of time, with the same colors,

again illustrating the boom-and-bust pattern. The panel on the right is a phase portrait of the data shown in the center; oscillations lead to cycles in the phase portrait and the initial large spike is seen as the large diameter cycle.

Conformational spread reactions

Currently, Smoldyn only allows second order reactions that have exactly two products to be declared a conformational spread reaction. Defining them as a conformational spread reaction, which is done with the `confspread_radius` statement, implies a few things. Typically, the diffusion coefficients of both reactants are zero, although this is not required. The reaction rate constant that is entered is a *first order* rate constant, meaning that it has units of inverse time. It is interpreted as the rate at which a reaction will occur, given that both reactants are continuously closer to each other than the conformational spread radius. Finally, the products of a conformational spread reaction are placed in the exact same locations as the reactants, and in the spots that correspond to the order in which the reactants and products were listed in the configuration file. For example, consider the conformational spread reaction defined with the statements

```
reaction rxn1 A + B -> C + D 10
confspread_radius rxn1 5
```

This states that a conformational spread reaction can occur between any A and B molecules that are closer than 5 distance units apart. At each time step, the probability of its occurring is found from the reaction rate of 10 inverse time units according to the same formulae that were described above for unimolecular reactions. If it occurs, the A molecule will be replaced by a C molecule and the B molecule will be replaced with a D molecule.

Conformational spread processes are frequently symmetric such that activity can be spread from an active molecule to its neighbor, and also inactivity can spread from an inactive molecule to its neighbor. This can be entered in Smoldyn with a pair of conformational spread reactions:

```
reaction rxna inactive + active -> active + active 10
reaction rxni active + inactive -> inactive + inactive 10
confspread_radius rxna 5
confspread_radius rxni 5
```

This will yield a warning in Smoldyn about there being multiple bimolecular reactions listed with the same reactants, but it is the right way to list these symmetric effects. In this example, the convention was followed that the latter reactant (and latter product) is the neighbor molecule, while the former reactant is the one that changes state.

If a molecule has simultaneous conformational spread interactions with more than one other molecule, the simulated reaction rates may be too low; this effect is reduced to zero for short time steps and increases with longer time steps. Consider a potential reaction with two reaction channels and the probability it happening by either channel

individually is p. When the two channels are considered sequentially, the probability for the first happening should be p, while the probability for the second should $p/(1-p)$, because it is the conditional probability of the second reaction happening, given that the first one did not happen. However, Smoldyn uses probability p for all conformational spread reaction channels, which leads to a reaction rate that is too low. While this identical effect is addressed correctly for first order reactions and for state conversions of surface-bound molecules, it is not addressed for conformational spread reactions because it is nearly impossible for Smoldyn to figure out how many reaction channels are available for any particular conformational spread reaction.

Conformational spread reactions were tested with the configuration file confspread.txt. It simulates two reactions:

```
reaction back green -> red 10
reaction fwd red + blue -> green + blue 10
confspread_radius fwd 5
```

While it is simplistic for most conformational spread situations, it leads to a simple equilibrium between red and green molecules which allows for easy analytical calculations of the correct outcome. If each red/green molecule is within a conformational spread radius of one blue molecule (accomplished by setting the conformational spread radius to 3), the forward and reverse rates are each 10 and an equal number of red and green molecules should be observed. On the other hand, an increased conformational spread radius (5, as shown above) implies that each red/green molecule is within reach of two blue molecules, so the forward rate doubles, as does the equilibrium constant. Both of these behaviors were confirmed. As described above, conformational spread reaction probabilities that were greater than about 0.05 for each reaction led to conformational spread reaction rates that were observed to be slightly too low for the case in which each red molecule was within the conformational spread radius of two blue molecules.

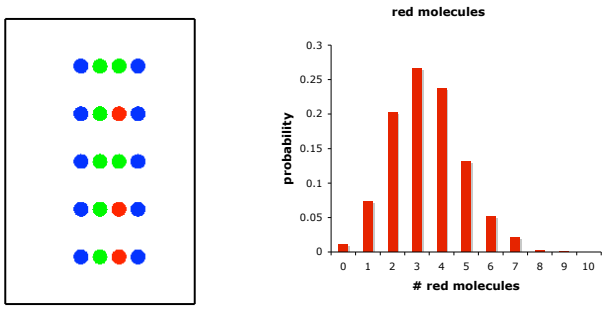

Figure 3.8.7: Output from confspread.txt configuration file. There are conformational spread reactions between blue molecules and red molecules, which convert red to green; reversion is a simple reaction. The panel on the right shows the average probability of molecules being in their red states, for a situation in which rate constants are equal for the forward and reverse reactions, but each red/green molecule is within a conformational spread radius of two blue molecules, thus doubling the red → green reaction rate.

Excluded volume reactions

Smoldyn does not treat molecules as though they truly have excluded volume because this would be a very computationally intensive procedure and it would require substantial rederivations of the reaction rate and surface interaction rate algorithms. However, using the same reaction concept that was developed for bimolecular reactions, Smoldyn does allow some excluded volume behaviors to be simulated, which are likely to be adequate for a wide range of simulations.

Essentially, the user specifies reaction binding and unbinding radii for a pair of species that are supposed to respect each others' excluded volume. If molecules of those two species end up within a binding radius of each other at the end of a time step, they "react", meaning that they are then moved apart to the unbinding radius. The reactants and products may be the same molecular species, in which case the molecules are simply pushed apart. Or, they may be different species, in which case an actual reaction occurs. The only substantial difference between normal bimolecular reactions and excluded volume reactions is that Smoldyn places the products of the former ones with random orientations about the reaction location, whereas those of the latter are placed along the same vector, and in the same sequence, as the reactants. The other difference is that the reaction rate value is essentially meaningless (and has not been calibrated) for excluded volume reactions.

If molecules are not supposed to pass by each other, which can be simulated using excluded volume reactions and a one-dimensional system, then it is important to make the excluded volume binding radius significantly larger than the rms step lengths of the molecules. Because molecules move during diffusion with Gaussian-distributed displacements, and Gaussians have long tails, it is likely to be very difficult to ensure that absolutely no molecules cross that should not.

We illustrate excluded volume reactions with two examples. In the former, called S8_reactions/bounce/bounce.txt, molecules are confined to a line and maintain their ordering. The configuration file statements that declare the excluded volume reactions are:

```
reaction rxn1 red(up) + green(up) -> red(up) + green(up)
binding_radius rxn1 1
product_placement rxn1 bounce 1.1
```

The second example involves a crowded system and is in the same directory and the file crowding.txt.

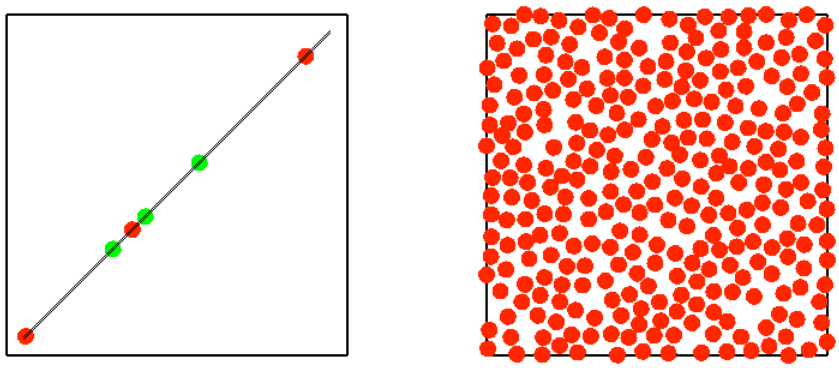

Figure 3.8.8: Output from bounce.txt and crowding.txt. In the former, red and green molecules, both of which are confined to the diagonal line, bounce off of each other. This has the result that the ordering of red and green molecules does not change during the simulation. The latter file shows that this crowding method works even with relatively high molecule densities. These molecules clearly do not overlap each other. During the simulation, molecules diffuse within the confines set by their neighbors.

Binding and unbinding radii

For every bimolecular reaction, Smoldyn has to calculate the correct binding radius from the reaction rate that is given in the configuration file. Also, for every reaction that leads to multiple products, Smoldyn has to determine the correct unbinding radius, using whatever product parameter is supplied, if any. Product parameters are listed in the table, below. While these binding and unbinding radii are well defined microscopic parameters (at least within the context of the Smoluchowski model system that is simulated), the meanings of the experimental rate constants, including those given in the configuration file, are not nearly as well defined. Instead, those rate constants depend on the conditions under which they were measured. Smoldyn accounts for this by attempting to guess the experimental conditions, using a process described here. If Smoldyn's guess is correct, the simulated reaction rates should exactly match the experimental rates (not including edge effects, which are typically negligible unless one reactant is fixed at or near an edge).

Table 3.8.2: Product parameters for reactions with multiple products

Special product types

`i`	`irrev`	reaction is declared irreversible (σ_u=0).
`a`	`confspread`	conformational spread reaction (entered automatically for you).

Use these if reversible reactions were measured at equilibrium

`p`	`pgem`	probability of geminate reaction (ϕ).
`x`	`pgemmax`	maximum probability of geminate reaction (ϕ_{max}).
`r`	`ratio`	unbinding radius relative to binding radius (σ_u/σ_b).

b	unbindrad	fixed length unbinding radius (σ_u).

Use these if reversible reactions were measured with all product removed as it was formed

q	pgem2	probability of geminate reaction (ϕ).
y	pgemmax2	maximum probability of geminate reaction (ϕ_{max}).
s	ratio2	unbinding radius relative to binding radius (σ_u/σ_b).
o	offset	fixed offset of products, rotationally randomized (σ_u).
f	fixed	fixed offset of products, not rotationally randomized (σ_u).

Either the single-letter code or the full word may be used to define the product parameter type, although the latter is suggested for readability. The default type is `pgemmax` with a value of 0.2.

In all cases, Smoldyn assumes that rate constants were measured using an effectively infinite number of reactant molecules, in an infinite volume, that were started well mixed and that then were allowed to react until either an equilibrium was reached for reversible reactions, or a steady-state reaction rate was reached for irreversible reactions. Only in these cases is mass action kinetics correct and is the reaction rate constant actually constant. The precise experimental assumptions are clarified with the following examples.

1. $A + B \rightarrow C$

   The rate constant is assumed to have been measured at steady state, starting with a well-mixed system of A and B. No product parameter is required. At steady-state, the simulation matches mass action kinetics.

2. $X \rightarrow A + B$

   There is no bimolecular reaction, so no binding radius is calculated. The default unbinding radius is 0, although it is possible to define a different one. If the product parameter type is `pgem`, `pgem2`, `ratio`, or `ratio2`, an error is returned due to the lack of a binding radius. If the parameter type is not given or is `irrev`, `pgemmax`, or `pgemmax2`, the unbinding radius is set to 0. If it is `unbindrad`, `fixed`, or `offset`, the requested separation is used. At steady-state, the simulation matches mass action kinetics.

3. $A + B \leftrightarrow C$

   If the reversible parameter is `pgem`, `pgemmax`, `unbindrad`, or `ratio`, the forward rate constant is assumed to have been measured using just this system of reactions after the system had reached equilibrium. The product parameter is used to yield the correct probability of geminate recombination if possible, or the desired unbinding

radius. In this case, the simulation matches mass action kinetics at equilibrium. If the product parameter is `pgem2`, `pgemmax2`, `ratio2`, `offset`, `fixed`, or `irrev`, then it is assumed that the forward rate constant was measured at steady-state and with all C removed as it was formed, thus preventing any geminate reactions. The unbinding radius is set as requested, using the binding radius if needed. In this case, the simulated forward reaction rate is higher than requested due to geminate rebindings.

4. $A + B \leftrightarrow C \rightarrow Y$

   The second reaction is ignored for determining parameters for A + B. Instead, the first reaction is considered as though the rates were determined experimentally using just the system given in example 3. If the product parameter is `pgem`, `pgemmax`, `ratio`, or `unbindrad`, the simulated reaction rate for the forward reaction $A + B \rightarrow C$ will be lower than the requested rate because there are fewer geminate reactions than there would be with the equilibrium system. Alternatively, it will be higher than the requested rate if the product parameter is `pgem2`, `pgemmax2`, `ratio2`, `offset`, `fixed`, or `irrev`, because there are some geminate reactions.

5. $X \rightarrow A + B \rightarrow C$

   The binding radius for the second reaction is treated as in example 1, without consideration of the first reaction. The unbinding radius for the first reaction is found using the binding radius of the second reaction. Here, product parameters `pgem` and `pgem2` are equivalent, `pgemmax` and `pgemmax2` are equivalent, and `ratio` and `ratio2` are equivalent. The actual reaction rate for the second reaction, found with a simulation, will be higher than the requested value due to geminate rebindings that occur after the dissociation of X molecules.

6. $X \rightarrow A + B \leftrightarrow C$

   The $A + B \leftrightarrow C$ binding and unbinding radii are treated as in example 3. Another unbinding radius is required for the first reaction, which is found as in example 5, using the binding radius from the second reaction. Mass action kinetics are not followed.

7. $X \leftrightarrow A + B \leftrightarrow C$

   The binding radii and unbinding radii for each bimolecular reaction are found as in example 3, independent of the other bimolecular reaction. The simulated rates may be different from those requested because of differing unbinding radii.

8. $X \rightarrow A + B \rightarrow C, \quad A + B \rightarrow D$

   The binding radii for the two bimolecular reactions are each found as in example 1. The unbinding radius for the first reaction cannot be determined uniquely, because the two forward reactions from A + B are equivalent and are likely to have different

binding radii. Smoldyn picks the binding radius for the first forward reaction that is listed. Thus, if the product parameter for dissociation of X is `pgem`, the requested geminate rebinding probability will be found for the reaction A + B → C, but a different value will be found for the reaction A + B → D.

9. C ↔ A + B ↔ C

This reaction scheme might represent two different pathways by which A and B can bind to form an identical complex. However, Smoldyn cannot tell which reverse reaction corresponds to which forwards reaction. Instead, for both determining the binding and unbinding radii, it uses the first reverse reaction that is listed.

The general principle for calculating binding radii is that Smoldyn first looks to see if a reaction is directly reversible (*i.e.* as in example 3, without any consideration of reaction network loops or other possible causes of geminate reactions). If it is and if the reversible parameter is `pgem`, `pgemmax`, `ratio`, or `unbindrad`, then the binding radius is found under the assumption that the rate constant was measured using just this reaction, at equilibrium. If not, or if the reversible parameter is `pgem2`, `pgemmax2`, `ratio2`, `offset`, `fixed`, or `irrev`, then Smoldyn calculates the binding radius with the assumption that the rate constant was measured using just that reaction at steady-state and with all product removed as it is formed.

Unbinding radii typically require a reversible parameter (except as in example 2). If the parameter is `unbindrad`, `offset`, or `fixed`, the requested unbinding radius is used. If it is `irrev`, the unbinding radius is set to 0. Otherwise, it can only be calculated with the knowledge of the binding radius. If the reaction is directly reversible, the binding radius for the reverse reaction is used. If it is not directly reversible but the products can react, as in examples 5, 6, and 8, then the binding radius for the first reaction that is listed is used.

Bimolecular reactions and surfaces

Does a bimolecular reaction occur if there is a surface between the reactants? This turns out to be a somewhat complex question. The simple answer is that it does occur if the surface is transparent to both molecular species and it does not occur if the surface is reflective or absorptive to both molecular species. In principle, reactions should be possible across pairs of jump surfaces, although they are not performed by the current Smoldyn version which treats jump surfaces as though they are opaque with respect to reactions.

Smoldyn determines where the reaction location is using a weighed average of the reactant diffusion coefficients. The reaction takes place only if both reactants can get to the reaction position, considering any intervening surfaces. Absorption on the opposite side of a surface is not worried about, the logic being that molecules are already in contact when a reactant traverses the surface, and so opposite-side absorption is no more important than the reaction. For partially transparent surfaces, reactions occur depending on the probability of transparency.

When molecules have excluded volume, which they do not in Smoldyn, even inert impermeable surfaces can affect the local concentrations of chemicals. An obvious effect is that a molecule cannot be closer to a surface than its radius, leading to a concentration of zero closer than that. In a mixture of large and small molecules, Brownian motion tends to push the large molecules up against surfaces while the small molecules occupy the center of the accessible volume, thus creating more complex concentration effects. These effects do not occur when excluded volume is ignored, as it is in Smoldyn, in which case surfaces do not affect local concentrations.

While surfaces do not affect concentrations of non-reacting molecules, they do affect reaction rates. Consider the reaction A + B $\rightarrow$ C, where A is fixed and B diffuses. If essentially all A molecules are far from a surface, the diffusion limited reaction rate is found by solving the diffusion equation for the radial diffusion function (RDF) with the boundary conditions that the RDF approaches 1 for large distances and is 0 at the binding radius (see the paper by myself and Dennis Bray titled "Stochastic simulation of chemical reactions with spatial resolution and single molecule detail"). This leads to the Smoluchowski rate equation

$$k = 4\pi D\sigma_b$$

However, for an A molecule that is near a surface, an additional boundary condition is that the gradient of the 3 dimensional RDF in a direction perpendicular to the surface is zero at the surface. This makes the solution of the reaction rate sufficiently difficult that I have not attempted to solve it, but the result is different from the simple result given above. This surface effect is an issue whenever the A molecule is within several binding radii of a surface and is especially pronounced when it is closer to the surface than its binding radius. For cases in which the A molecule is more than one binding radius from the surface, B molecules are going to take longer than usual to reach the region between the A and the surface, leading to a decreased reaction rate. It is suspected that the reaction rate decreases monotonically as the A molecule approaches and then crosses a surface.

A special case that can be solved exactly occurs when the A molecule is exactly at the surface, such that half of the binding volume is accessible to B molecules and half is inaccessible. Now, the RDF inside the system volume is identical to the RDF for the case when the A molecule is far from a surface. The logic is to assume that this is true and to then observe that it already satisfies the additional boundary condition. Using this RDF, the diffusive flux is half of the diffusive flux for an A molecule far from a surface, because only half of the binding surface is exposed to the system. Thus, the diffusion limited reaction rate for the situation in which a reactant is fixed exactly at a surface is

$$k = 2\pi D\sigma_b$$

The situation changes some when simulation time steps are sufficiently long that rms step lengths are much longer than binding radii. Now, the probability of a reaction occurring during a time step is a function of only the binding volume. Thus, there are no surface effects at all when an A molecule is fixed anywhere in the simulation volume that is greater than or equal to one binding radius away from a surface. As the A molecule is

moved closer to the surface, the reaction rate decreases in direct proportion to the binding volume that is made inaccessible to B molecules. An especially easy situation is that when the A molecule is exactly at the surface, the reaction rate is half of its value when the A molecule is far from a surface, which is the same as the diffusion limited result.

These results can be turned around to solve for the binding radius. If the reaction is diffusion limited, the binding radius should double when a reactant is placed exactly at the surface to maintain the same reaction rate. If it is activation limited, the binding radius should increase by $2^{1/3}$ to maintain the same reaction rate. As usual though, the binding radius is more closely related to the fundamental physical properties of the molecule than is the rate constant, so it is essential to consider the experimental conditions that were used for measuring the rate constant.

In conclusion, reaction rates are reduced near surfaces and the effect is different for diffusion limited and activation limited reactions. However, for both cases, and almost certainly for all cases in between, the reaction rate is exactly half when an A molecule is fixed at a surface, compared to when it is far from a surface. A few tests with Smoldyn using the files wallreact.txt, suggested that these surface effects are likely to be minimal for most situations, although it is good to be aware of their potential. The exception is that there are large surface effects when molecules are fixed with a significant portion of the binding volume outside the simulation volume.

3.9 Compartments

Compartment basics

Compartments are regions of volume that are bounded by surfaces. They do not include their bounding surfaces. Compartments are useful for input or output and, as mentioned above, zeroth and first order reactions can be made to be only active within specified compartments. Also, they are used for communication with the MOOSE simulator.

The inside of a compartment is defined to be all points from which one can draw a straight line to one of the "inside-defining points" without crossing any bounding surface. For example, to create a spherical compartment, one would define a spherical surface as the boundary and some point inside the sphere (the center, or any other internal point) to be the inside-defining point. This definition allows a wide variety of options. For example, it allows disjoint compartments and compartments that are not inside closed surfaces. To set a sharp edge to a compartment, but one which does not affect molecule diffusion, just add a surface that is transparent to all molecules but which serves as one of the compartment's bounding surfaces.

In addition, compartments can be composed from previously defined compartments using logic arguments. This way, for example, a cell cytoplasm compartment can be defined as the region that is within a cell compartment but that is not also within a nucleus compartment. Or, the region that is outside of a cell can be simply defined as the region that is not inside the cell.

Defining compartments

The definition style for compartments is much like it is for other portions of the code. Compartment statements for specific compartments are entered in blocks that start with `start_compartment` and end with `end_compartment`. The compartment name, which is given after `start_compartment`, is used to start a new compartment definition, or to continue defining a previously started one. Bounding surfaces and interior-defining points are added with the `surface` and `point` statements, respectively. The `compartment` command, used within a compartment block, is used to define one compartment in terms of others. Using this command one can, for example, define a compartment as the union or the intersection of two previously defined compartments.

To state that molecules start in a compartment, use the `compartment_mol` statement that was listed in the molecules section. To read the numbers of molecules in a compartment, use the command `molcountincmpt` or `molcountincmpt2`.

Following are excerpts from configuration files that use compartments:

Compartment defined with surfaces and points

```
start_compartment middle
surface surf
point 50 75
point 50 25
point 75 50
point 25 50
```

```
end_compartment

compartment_mol 500 red middle
```

Compartments defined with other compartments

```
start_compartment intersection
compartment equal left
compartment and right
end_compartment

start_compartment either
compartment equal left
compartment xor right
end_compartment

start_compartment outside
compartment equalnot left
compartment andnot right
end_compartment

compartment_mol 500 red intersection
compartment_mol 500 green either
compartment_mol 500 blue outside
```

These files are in the examples folder in S9_compartments. The first is called compart.txt and the second is compartlogic.txt. They yield the following results:

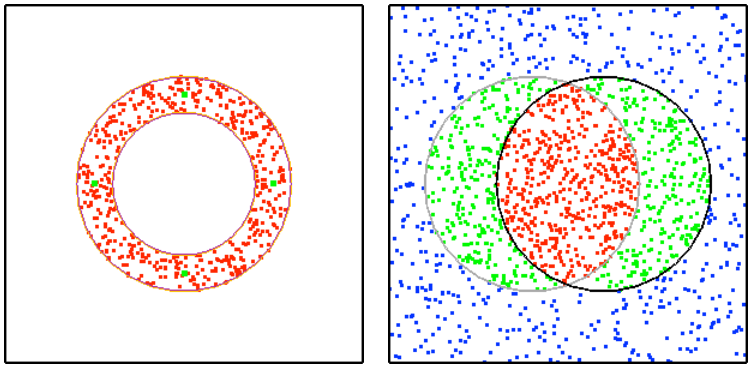

Figure 3.9.1: Examples of compartments. In the left panel, green dots are the interior-defining points and red molecules were added randomly to the compartment. In the right panel, each circle was defined as a compartment and then the red, green, and blue molecule regions were defined with logical combinations of the left and right compartments.

For logically combining compartments, the logical options are: “equal”, “equalnot”, “and”, “andnot”, “or”, “ornot”, or “xor”. These obey the standard logical rules. Note that the sequence of statements matters. For example, the region defined by A-andnot-B is the portion of A is that is not within B, whereas B-andnot-A is the portion of B that is not within A.

Table 3.9.1: statements about compartments

`max_compartment` *int* (optional statement)
`start_compartment` *name*
`surface` *surface*
`point` $pos_0 \ldots pos_{dim-1}$
`compartment` *logic compart*
`end_compartment`

3.10 Simulation settings

Simulation settings basics

Several statements define how the simulation should be run. There are defaults for each of these settings, so the user does not need to set them directly. However, they can be useful for optimizing simulation performance. These settings include the random number generator seed, virtual boxes that partition the simulation volume, and some settings for diffusion on surfaces.

The simulation volume is partitioned into an array of virtual boxes, each of which is the same size and shape. In addition, each box that is on the edge of the simulation volume actually extends out to infinity in that direction, such that every location in space, whether in the simulation volume or not, is in some virtual box. These boxes do not affect the performance of the simulation, except for allowing computational efficiencies that speed it up.

Random number seed

As a default, the random number generator seed is set to the time at which the simulation is started. This is virtually certain to yield a unique random number sequence each time the simulation is run, so no two simulations will be identical. However, it can also be useful (primarily for code debugging) to set the random number generator seed, which can be done with the `random_seed statement`.

Virtual boxes

The box sizes can be left undefined, in which case a default is used, or they can be defined with either the `molperbox` or `boxsize` statements. The former statement sets the box sizes so that the average number of molecules per box, at simulation initiation, is close to the requested number. Good numbers tend to be between 3 and 6, although more or fewer may be appropriate, depending on how the number of molecules in the simulation is likely to change over time (the default box size is computed for an average of 4 molecules per box). The `boxsize` statement requests the length of one side of a box, which should be in the same units that are used for the boundary statements. Either way, the boxes that are actually created are unlikely to exactly match the requested values, but are sized to be as close to cubical as possible (or square for a 2-D simulation) and to exactly fill the simulation volume.

Box sizes that are too large will cause slow simulations, but no errors. Warnings that say that there are a lot of molecules or surface panels in a box are suggestions that smaller boxes may make the simulation run faster, but do not need to be heeded. Box sizes that are too small may cause errors. Several warnings can be generated for this, including that the diffusive step lengths are larger than the box size, etc. However, the only warning that really matters is if box sizes are smaller than the largest bimolecular reaction binding radius. If this happens, some bimolecular reactions are likely to be ignored, which will lead to a too slow reaction rate. If simulation speed is important, it is

a good idea to run a few trial simulations with different box sizes to see which one leads to the fastest simulations.

The `accuracy` statement sets which neighboring boxes are checked for potential bimolecular reactions. Consider the reaction A + B → C and suppose that A and B are within a binding radius of each other. This reaction will always be performed if A and B are in the same virtual box. If accuracy is set to at least 3, then it will also occur if A and B are in nearest-neighbor virtual boxes. If it is at least 7, then the reaction will happen if they are in nearest-neighbor boxes that are separated by periodic boundary conditions. And if it is 9 or 10, then all edge and corner boxes are checked for reactions, which means that no potential reactions are overlooked. Overall, increasing accuracy numbers lead to improved quantitative bimolecular reaction rates, along with substantially slower simulations. If qualitative simulations are wanted, then lower accuracy values are likely to be preferable.

Surface-bound molecule settings

Several settings affect simulation of surface-bound molecules, described here. The default settings are nearly always good, although they can be modified if desired.

Molecules that are bound to a surface are given locations that are extremely close to that surface. However, this position does not need to be exactly at the surface, and in fact it usually cannot be exactly at the surface due to round-off error. The tolerance for how far a surface-bound molecule is allowed to be away from the surface can be set with the `epsilon` statement.

When a surface-bound molecule diffuses off of one surface panel, it can sometimes diffuse onto the neighboring surface panel. It does so only if the neighboring panel is declared to be a neighbor, as described above in the surfaces section, and also the neighbor is within a distance that is set with the `neighbor_dist` statement. This value is set to an extremely small value by default, just large enough to prevent round-off error, and generally should not need changing. In some cases, moving a molecule to a point that is exactly on a panel edge can cause problems with round-off errors, so it is actually moved just inside the edge by a distance that can be set by the `margin` statement. Again, this should not need changing.

Table 3.10.1: statements for simulation settings

`random_seed` *int*	random number seed
`accuracy` *float*	accuracy code, from 0 to 10
`molperbox` *float*	target molecules per virtual box
`boxsize` *float*	target size of virtual boxes
`epsilon` *float*	for surface-bound molecules
`margin` *float*	for diffusing surface-bound molecules
`neighbor_dist` *float*	for diffusing surface-bound molecules

3.11 Ports

Port basics

Ports are data structures that are used for importing and exporting molecules between a Smoldyn simulation and another simulation. In particular, they are designed for the incorporation of Smoldyn into MOOSE, but they could also be used to connect multiple Smoldyn simulations or for other connections.

A port is essentially a surface and a buffer. Smoldyn molecules that hit the porting surface are removed from the Smoldyn simulation and are put into the buffer for export. Once exported, they are removed from the buffer. Also, molecules may be added to the Smoldyn simulation at the porting surface by other programs.

Defining ports

Using the standard format, port statements are given in blocks that start with `start_port` and end with `end_port`. A port name is declared after `start_port`. The porting surface is specified with `surface` and the active face of that surface is specified with `face`.

Also, in the definition of the surface that is to be used for porting (the surface has to be defined first), one has to specify that the active face of the surface has action "port".

Table 3.11.1: statements about ports

`start_port` *name*
`surface` *surface*
`face` *face*
`end_port`

Porting rate

Some care is required to make ports work accurately. In particular, a port behaves for a Smoldyn simulation as an absorbing surface. The absorption rate depends on the simulation time step and molecular rms step length, as I described in Andrews, *Physical Biology*, 2009.

3.12 Rule-based network generation

Smoldyn included the Libmoleculizer module for rule-based network generation up to version 2.31. The software was almost very useful, but not quite. Nothing ever worked quite as desired, and the module was very difficult to compile. Thus, it was completely removed, starting at version 2.32. Version 2.34 has a nearly complete translation module that translates BioNetGen output into Smoldyn input. However, it still needs validation and documentation. If you need these features, let us know and we'll try to speed up the finalization.

Here is the current situation, as of version 2.37:

The BioNetGen integration with Smoldyn is only half complete. However, it should still be workable with full functionality, just a little harder to use than it should be.

The current functionality is shown with the example file blbr.bng, which is in the Smoldyn distribution in examples/S12_bionetgen/blbr/ (and also in the BioNetGen distribution). The first step is to use BioNetGen to generate a .net file. At a command line, I entered the following statement:

```
../../../source/BioNetGen/BNG2.pl blbr.bngl
```

This ran the perl program BNG2.pl, which is in my source directory, which read the file blbr.bngl and produced blbr.net. BNG2.pl is not standard in the source directory, so you will probably need to navigate to a different place to run it (you also may need to download it from the BioNetGen site).

Next, run Smoldyn using the file blbrsim.txt, which should be in the examples/S12_bionetgen/blbr/ directory. If this fails with a syntax error it's probably because I changed the syntax so that the line which says "start_bngnet blbr" should be rewritten as "start_bng blbr". After making that change in the blbrsim.txt file, it should run. This file seems to run exceedingly slowly, although it's not clear why.

Following this scheme, it should be fairly easy to do the same thing with other files. It's generally advisable to start with small simple networks at first, and read all of the intermediate files and the Smoldyn text output, just to make sure that things are working as intended. Once files get big, you will just have to trust that they are working right, so it's nice to verify that things are working correctly first.

3.13 Filaments

I am working on adding simulation support for filaments to Smoldyn, but have only just begun. At present, it is possible to define filaments and specify their geometries by adding monomers to them. These filaments can move by treadmilling, and they interact with surfaces. They do not exhibit Brownian motion. See the examples in the `S13_filaments` directory.

3.14 Hybrid simulation with adjacent lattices

Most of the Smoldyn software is developed around the Smoluchowski level of detail. Here, each individual molecule of interest in the simulation is represented as a small sphere that has a precisely defined position in continuous space. This offers spatial accuracy down to nanometer size scales for typical systems, which is more detailed than that offered by most other comparable simulation software, but is necessary when studying biophysical processes that take place on these spatial scales. The cost of this high level of detail is that simulations become computationally demanding, both in terms of the number of processes that have to be run at each simulation time step and in terms of the memory required to store all of the molecular information. Hybrid simulations can offer solutions for simulating models with both high levels of detail and high speed, which they accomplish by representing high levels of detail only as needed.
The hybrid methods that are particularly important here combine particle-based simulation with lattice-based simulation. The particle-based simulation methods are Smoldyn's standard methods, which work at the Smoluchowski level of detail. The lattice-based methods represent spatially compartmentalized versions of the chemical master equation, typically simulated using one of the spatial Gillespie methods (partial differential equations or spatial Langevin methods are also appropriate). Hybrid methods can use either overlapping space or adjacent space methods. In the former case, the physical space represented by the lattice-based methods is the same as that by the particle-based methods; molecules in one representation can interact with spatially proximate molecules that are in the other representation. Smoldyn has been added to Virtual Cell in this manner, where VCell provides the lattice representation and Smoldyn provides the particle representation. Here, the lattice representation is best for abundant or rapidly diffusing species where exact molecule positions don't matter, and the particle representation for rare species where the extra computational effort is necessary. In the latter case, the particle-based and lattice-based methods represent adjacent regions of physical space. Molecules can diffuse back and forth between the two regions, changing representations as they do so. This approach is best in cases where one region of space needs to be simulated in detail, while surrounding regions can be simulated more coarsely. The remainder of this section focuses on this latter adjacent space approach.

Lattices were a new feature in Smoldyn 2.32.

Hybrid simulation basics

The lattice module incorporated into Smoldyn is fairly simple. It represents lattices using an axis-aligned rectangular array of subvolumes. It simulates chemical reactions using the next subvolume (NSV) method, which treats molecules as discrete objects (i.e. not continuously variable concentrations) and captures reaction stochasticity accurately. Whereas simulation time advances with fixed length time steps in the particle-based methods, it advances with unequal steps, from event to event, in the NSV method. The lattice region of space can be bounded by a few different boundary types, but the lattice code does not currently address interactions between molecules and any surfaces that are

within the lattice region of space. The junction between the particle-based region of space and the lattice region of space is created using a Smoldyn "port", explained above.

Defining lattices

To include a lattice in a model, you need to add a lattice, obviously. This is entered using a block of statements that starts with `start_lattice` and ends with `end_lattice`, much like similar blocks for surfaces, compartments, and other things. The definitions that can be entered within this block are discussed below. In addition to adding a lattice, you also need to define a port, which will form the junction between the particle space and the lattice space. And to create a port, you will need to define at least one surface. The `examples/S14_lattices/diffusion.txt` file shows a very simple example of model that uses a lattice.

First, it's a good idea to define the lattice type using the `type` statement. In principle, this will enable you to choose whether the lattice region is simulated with discrete numbers of molecules using NSV algorithm, with continuous concentrations using PDE algorithms, or with other methods. In practice though, only NSV is currently implemented, and NSV is the default, so you don't actually need to define the type. On the other hand, you do need to define the port that separates particle space from lattice space, using the `port` statement.

Define the boundaries of the lattice space using the `boundaries` statement. It is essentially identical to the `boundaries` statement for the main portion of the configuration file, but that one only applies to the particle region of space and this one only applies to the lattice region of space. The two sets of boundaries are typically strictly adjacent to each other, with no gap and no overlap, but it is also just fine if they overlap. The port should obviously be at the intersection of the two sets of boundaries, or somewhere within the overlap region. By default the lattice boundaries are reflective, but they can also be periodic. These are entered with optional characters after the rest of the statement, exactly as for the particle side `boundaries` statement.

Lattice partitioning is defined using the `lengthscale` statement. Typically, the values entered here are an even divisor of the `boundaries` dimensions. I'm not sure what happens if this is not true.

Use the `species` and `reactions` statements to tell a lattice which species and reactions it should work with. Often, "`all`" is used, meaning that the lattice should know about all of the same species and/or reactions as the particle side of the simulation uses. However, it's also possible to specify a subset of the total species and reactions lists. This is useful because the lattice code runtime increases with more species and with more reactions, unlike the particle side, which increases with numbers of individual molecules. Lattices cannot work with any species or reactions that are not also defined in the particle side. However, it is possible to have a reaction only perform on the lattice side. In this case, define the reaction on the particle side, with a rate constant as usual. Then, when listing the reactions that the lattice side should work with, use the keyword "`move`" to indicate that all subsequent reactions in the list should be "moved" to the lattice side and disabled on the particle side.

Finally, use the `mol` statement to add molecules to the lattice side. This is essentially identical to the statement of the same name in the main portion of the configuration file, but only applies to the lattice side of space.

Lattice output

Several commands output information from lattices. `printLattice` outputs some basic information about the lattice, including the low and high corners of the lattice space, the subvolume partition spacing, and the total number of each species in the lattice. This is the same output that is displayed with the simulation diagnostics.

`molcount` and `molcountspace` are functions that are often used with non-lattice simulations. In addition to counting molecules in the particle region of space, they also count molecules in the lattice region; there is no way to select just particle region or just lattice region. `molcountspace` does not count molecules that are in transit between representations (if you select a single species and state; it does if you select all species and/or all states), so it will miss a few molecules. `savesim` saves the full simulation state; it saves the lattice state as well as the rest. Other molecule counting commands do not include lattice molecules.

Finally, `writeVTK` produces VTK output for both the particle and lattice regions of space. It does not include surface information. The output is saved as a stack of files that have names that follow the format *filename*`Lattice00_00001.vtu` and *filename*`Molecules00001.vtu`, and that have incremented numbers for subsequent snapshots. This output can be viewed using Paraview, Visit, or other VTK viewers. It doesn't appear that any of them are trivial to use.

Table 3.13.1 statements about lattices

`start_lattice` *name*	start defining a lattice
`type` *type*	type of the lattice ("nsv")
`port` *port*	port for exchanging molecules
`boundaries` *dim pos1 pos2 type*	boundaries of the lattice region of space
`lengthscale` *x1 x2 x3*	partition spacing for lattice subvolumes
`species` *species1 species2 ...*	species that the lattice should recognize
`reaction [move]` *reaction1 reaction2 ...*	reactions that the lattice should recognize
`mol` *nmol name* $pos_0\ pos_1 \ldots pos_{dim-1}$	starting molecules in the lattice space
`end_lattice`	end the lattice block

4. Reference: statements and commands

4.1 Configuration file statements

Statements about the configuration file

`#` *text*

Single-line comment. A '`#`' symbol indicates that the rest of the line is a comment.

`/*`
text
`*/`

Multi-line comment. All lines between "/*" and the following "*/" are ignored. These must be the first "words" on a line. Additional text on these lines is ignored as well. In future versions, the syntax of these may be changed so as to be identical to C-style block comments.

`read_file` *filename*

Read some other configuration file, returning to the present one when that one has been read.

`end_file`

End of configuration file. This line is optional (but good programming practice), as Smoldyn can also just read until the file ends.

`quit_at_end` *value*

Use a value of "yes" to tell the simulator to quit the program at the end of the simulation, during simulations that use graphics. This has no effect if simulations do not use graphics. Use "no" to turn this off, which is the default behavior.

`define` *key substitution*

Definition of macro replacement text. Throughout the remainder of this configuration file, but not files that are called by it, all incidents of the string *key* are replaced with the string *substitution* before further parsing is performed. It is permissible to not include any substitution text.

`define_global` *key substitution*

Definition of macro replacement text, which is identical to `define`, except that this definition applies throughout both this file and all files that are called by it. Global definitions can also be entered on the command line using the `--define` option.

`undefine` *key*

Removes a macro substitution definition that was made previously, whether global or local. Global undefines apply to this file and all files that are called by it, but not to a file that called this one. Entering *key* as "all" undefines all definitions.

`ifdefine` *key*

The following lines of the configuration file are read only if *key* is a term that was defined with `define` or `define_global` (or was defined automatically, which includes `FILEROOT`). Reading, or not reading, continues to any `else` statement. The end of the condition is given with the `endif` statement.

`ifundefine` *key*

This is identical to `ifdefine`, except that reading continues only if *key* has not been defined.

`else`

This is the else condition which is supposed to follow an `ifdefine` or `ifundefine` statement.

`endif`

This ends a condition that is started by an `ifdefine` or `ifundefine` statement.

`display_define`

Causes all current definitions to be displayed to the standard output. This is only useful for debugging define issues in configuration files.

Statements about space and time

`dim` *dim*

Dimensionality of the system. Must be at least one, and is typically between 1 and 3. Larger numbers are permitted as well.

`boundaries` *dim pos1 pos2*
`boundaries` *dim pos1 pos2 type*

Creates lower and upper boundaries to define the simulation volume on dimension *dim*. The *dim* value should be 'x', 'y', or 'z' (however, 0, 1, and 2 work as well). These boundaries are located at *pos1* and *pos2*. Using the first format, which is advised for systems that include surfaces, boundaries are created that are transparent to molecules, meaning that they do not contain or otherwise interact with molecules. Surfaces need to be defined to keep molecules in the system. The second format is preferable for systems that do not include any surfaces. In this case, the boundary type can be 'r' for reflective, 't' for transparent, 'a' for absorbing, or 'p'

for periodic. For most purposes, this statement replaces the `low_wall` and `high_wall` statements.

`low_wall` *dim pos type*

This statement has been largely superseded by boundaries. This creates a lower boundary for the simulation volume. This wall is perpendicular to the dimension *dim* ('x', 'y', or 'z') such that all locations between *pos* and the position of the corresponding upper boundary are considered to be within the simulation volume. The type of wall is given in *type*, which should be one of four single letter codes: '`r`' means a reflecting wall, '`p`' means a periodic wall (also called wrap-around or toroidal), '`a`' means an absorbing wall, and '`t`' means a transparent wall. Transparent walls imply an unbounded system and may lead to slow simulations. *If any surfaces are defined for the simulation, then walls still must be entered to define the system volume, but these walls are essentially non-functional (the sole exception is that reactions can occur across periodic walls). Additional surfaces need to be defined to serve as the system boundaries.*

`high_wall` *dim pos type*

This statement has been largely superseded by boundaries. This is identical to the definition for `low_wall`, although this creates the upper boundary for the simulation volume. See note about surfaces in `low_wall`.

`time_start` *time*

Starting point for simulated time.

`time_stop` *time*

Stopping time of simulation, using simulated time. The simulation continues past the `time_stop` value by less than one time step.

`time_step` *time*

Time step for the simulation. Longer values lead to a faster runtime, while shorter values lead to higher accuracy. Also, longer values lead to bimolecular reactions that behave more as though they are activation limited, rather then diffusion limited.

`time_now` *time*

Another starting time of simulation. Default value is equal to `time_start`. If this time is before `time_start`, the simulation starts at `time_start`; otherwise, it starts at `time_now`.

Statements about molecules

`species` $name_1$ $name_2$ … $name_n$

Names of one of more molecular species present in the system. Standard naming conventions are followed, in that the name should start with a letter and spaces are not permitted.

`difc` *name value*
`difc` *name*(*state*) *value*

Isotropic diffusion coefficient of molecule type *name*. Default value is 0. The state, which is optional, refers to the surface-bound state of the molecule: solution, front, back, up, or down; if omitted, only the solution state is set with this statement. *name* may be "all" and/or *state* may be "all" to set diffusion coefficients for multiple species at once.

`difm` *name* $float_0$ $float_1$ … $float_{dim*dim-1}$
`difm` *name*(*state*) $float_0$ $float_1$ … $float_{dim*dim-1}$

Square root of diffusion matrix of *name* and maybe state *state* (the dot product of this matrix and itself is the anisotropic diffusion matrix). The matrix has dim^2 terms (*dim* is the system dimensionality), listed row by row of the matrix; the matrix is supposed to be symmetric. If this line is not entered, isotropic diffusion is assumed, which leads to a faster runtime. While a matrix is used for diffusion if one is given, the value stored with `difc` is used for reaction rate calculations. If `difc` is not entered, the trace of the square of this matrix, divided by the system dimensionality, is used as a proxy for the isotropic diffusion coefficient to allow reaction rates to be estimated. This line is most useful for restricting diffusion to a plane or a line, in which case the square root of the diffusion coefficient is given for each diagonal element of the matrix where there is diffusion and 0s are place on diagonal elements for axes where diffusion is not possible, as well as on off-diagonal elements. *name* and or *state* may be "all" to set diffusion matrices for multiple species at once.

`drift` *species* $float_0$ $float_1$ … $float_{dim-1}$
`drift` *species*(*state*) $float_0$ $float_1$ … $float_{dim-1}$

Drift velocity vector for molecules of type *species* and maybe state *state*. The vector has *dim* terms (*dim* is the system dimensionality). If this line is not entered, there is no net drift. *species* and or *state* may be "all" to set drift vectors for multiple species at once.

`surface_drift` *species*(*state*) *surface panel-shape* $float_0$ … $float_{dim-2}$

Drift velocity vector for molecules of type *species* and state *state*, relative to the local coordinates of the panel to which these molecules are bound. The vector has *dim*-1 terms (*dim* is the system dimensionality), which are for the natural coordinate system of the local panel. *species* and or *state* may be "all" to set drift vectors for multiple species and surface-bound states at once.

`mol` *nmol name* pos_0 pos_1 … pos_{dim-1}

Simulation starts with *nmol* type *name* molecules at location *pos*. Each of the *dim* elements of the position may be a number to give the actual position of the molecule or molecules; or the letter 'u' to indicate that the position for each molecule should be a random value between the bounding walls, chosen from a uniform density; or a position range which is given as two numbers separated with a hyphen.

`surface_mol` *nmol species*(*state*) *surface pshape panel* $pos_0\ pos_1 \ldots pos_{dim-1}$
`surface_mol` *nmol species*(*state*) *surface pshape panel*

Creates surface-bound molecules. *nmol* molecules of type *species* are created on the surface named *surface*, on the panel with shape *pshape* and name *panel*. They are all put in state *state*, which can be "front", "back", "up", or "down". If additional text is entered, it needs to be the Cartesian coordinates of the molecules, all of which are put at the same spot and on the same panel. If the coordinates are not given, the molecules are placed randomly on the surface with a constant density, on average. For randomly placed molecules, it is permissible to enter "all" for the panel, the *pshape*, and/or the surface.

`compartment_mol` *nmol species compartment*

Creates *nmol* solution-phase molecules of type *species* in the compartment named *compartment*.

`molecule_lists` $listname_1\ listname_2 \ldots$

Creates and names a set of molecule lists, for molecules that are in the system. This statement may be called multiple times.

`mol_list` *species listname*
`mol_list` *species*(*state*) *listname*

Assigns all molecules that are in the system and of type *species* and state *state* (if *state* is not specified, then only the solution state is assigned) to the list called *listname*.

`max_mol` *int*

Optional statement (it was required up to version 2.22). This tells Smoldyn to terminate if more than this many molecules end up being used for the simulation.

Statements about graphics

`graphics` *str*

Type of graphics to use during the simulation. The options are '`none`' for no graphics, '`opengl`' for basic and fast OpenGL graphics, '`opengl_good`' for fair quality OpenGL graphics, and '`opengl_better`' for pretty good graphics. Runtime gets slower with better quality. If this line is not entered, no graphics are shown.

`graphic_iter` *int*
: Number of time steps that should be run between each update of the graphics. Default value is 1.

`graphic_delay` *float*
: Minimum amount of time in milliseconds that Smoldyn should pause between successive graphics updates. Default is 0.

`quit_at_end` *yes/no*
: Whether Smoldyn should quit running as soon as the simulation is complete or not. Enter `yes` (or 1) if it should and `no` (or 0) if not.

`frame_thickness` *int*
: Thickness of the frame that is drawn around the simulation volume, in points. Default value is 2.

`frame_color` *color* [*alpha*]
`frame_color` *red green blue* [*alpha*]
: Color of the frame. All values should be between 0 and 1; use all 0s for black and all 1s for white (default). The *alpha* value is optional and also useless.

`grid_thickness` *int*
: Thickness of the grid lines that can be drawn to show the virtual boxes. Default value is 0, so that the grid is not drawn.

`grid_color` *color* [*alpha*]
`grid_color` *red green blue* [*alpha*]
: Color of the grid. All values should be between 0 and 1; use all 0s for black and all 1s for white (default). The *alpha* value is optional and also useless.

`background_color` *color* [*alpha*]
`background_color` *red green blue* [*alpha*]
: Color of the background. All values should be between 0 and 1; use all 0s for black and all 1s for white (default). The *alpha* value is optional and may not work anyhow.

`display_size` *name float*
`display_size` *name*(*state*) *float*
: Size of molecule of type *name* for display to the graphical output. If the surface state is omitted, as in the first form shown, this display size applies to all molecule states; otherwise it applies to only the state listed. These states may be “solution”, “front”, “back”, “up”, “down”, or “all”. The default value is 3, indicating that each molecule is displayed with a small

square; 0 indicates that a molecule should not be displayed and larger numbers yield larger squares.

`color` *name*(*state*) color [*alpha*]
`color` *name*(*state*) *red green blue* [*alpha*]
Color for displaying molecules of type *name*. If the surface state is omitted, this color applies to just the solution state. States may be "solution", "front", "back", "up", "down", or "all". Colors can be words, or can be given with red, green, and blue values, each of which should be between 0 and 1. Default values are 0 for each parameter, which is black. Entering *alpha* is optional and useless.

`tiff_iter` *int*
Number of time steps that should be run between each automatic saving of a TIFF file. Default value is 0, meaning that TIFFs should not be saved automatically.

`tiff_name` *name*
Root filename for TIFF files, which may include path information if desired. Default is "`OpenGL`", which leads to the first TIFF being saved as "`OpenGL001.tif`".

`tiff_min` *int*
Initial suffix number of TIFF files that are saved. Default value is 1.

`tiff_max` *int*
Largest possible suffix number of TIFF files that are saved. Once this value has been reached, additional TIFFs cannot be saved. Default value is 999.

`light` *number parameter color* [$value_4$]
`light` *number parameter* $value_1$ $value_2$ $value_3$ [$value_4$]
Set the parameters for a light source, for use with opengl_better quality graphics. The light *number* should be between 0 and 7. The *parameter* may be one of four strings: "ambient", "diffuse", "specular", or "position". The first three parameters are for the light's colors, which are then specified with either a word or in the *value*s as red, green, blue, and optionally alpha. The last parameter type is for the light's 3-dimensional position, which is specified as *x*, *y*, and *z* in the *value*s. Lights specified this way are automatically enabled (turned on).

`text_color` *color*
`text_color` *red green blue*
Color for text displayed on the graphics window.

`text_display` $item_1$ $item_2$ ...

Turns on text display of the listed items, which are listed as strings. Possible items are "time", which is the simulation time, and species names and states (entered as species(state)), for which the number of molecules of that species and state are displayed. Wildcards are permitted.

Statements about run-time commands

`output_root` *str*

Root of path where text output should be saved. Spaces are permitted. Output files are saved in the same folder as the configuration file, modified by this string. See the description for `output_files`. Make sure that the destination folder has been created and that the string is terminated with a colon (and started with a colon if needed).

`output_files` str_1 str_2 ... str_n

Declaration of filenames that can be used for output of simulation results. Spaces are not permitted in these names. Any previous files with these names will be overwritten. The path for these filenames starts from the configuration file and may be modified by a root given with `output_root`. For example, if the configuration file was called with `folder/config.txt` and `output_root` was not used, then the output file `out.txt` will appear in the folder `folder` too. If the configuration file was called with `folder/config.txt` and the output root was given as `results/`, then the output file goes to the `results` sub-folder of the folder `folder`. The filename "`stdout`" results in output being sent to the standard output (this does not need to be declared with the `output_files` statement).

`output_precision` *int*

The precision that will be used for numerical output from commands, meaning the number of digits displayed after a decimal point. Enter a negative number for the default and a positive number for fixed precision. For example, if you enter 5, then the output format string will be '`%.5g`'.

`append_files` str_1 str_2 ... str_n

Identical to `output_file`, except that the prior contents of these files are not overwritten, but are appended to.

`output_file_number` *int*

Starting number of output file name. The default is 0, meaning that no number is appended to a name (*e.g.* the file name `out.txt` is saved as `out.txt`). A value larger than 0 leads to an appended file name (if 1 is used, then `out.txt` is actually saved as `out_001.txt`). Note that the command `incrementfile` increments the file number before it runs the rest of the command.

`cmd b,a,e` *string*
`cmd @` *time string*

cmd n *int string*
cmd i *on off dt string*
cmd j *onit offit dtit string*
cmd x *on off dt xt string*

Declaration of a command to be run by the run-time interpreter, where the final portion labeled *string* is the actual command. The character following cmd is the command type, which may be 'b' for before the simulation, 'a' for after the simulation, 'e' for every time step during the simulation, '@' for a single command execution at time *time*, 'n' for every *n*'th iteration of the simulation, 'i' for a fixed time interval, 'x' for a fixed time multiplier, or 'j' for every *dtit* step with a set starting iteration and stopping iteration. For type 'i', the command is executed over the period from *on* to *off* with intervals of at least *dt* (the actual intervals will only end at the times of simulation time steps). For type 'x', the command is executed at on, then on+dt, then on+dt*xt, then $on+dt*xt^2$, and so forth. See section 2.4 for the commands that are available.

max_cmd *int* (obsolete statement)

Maximum length of command queue. Default value is 10. As of version 1.55, this statement is no longer needed in configuration files, because the command queue is now expanded as needed.

Statements about surfaces

The statements shown below that are preceded by an asterisk need to be entered within surface blocks, which start with start_surface and end with end_surface. These statements can also be entered directly, meaning not in a surface block, by preceding the statement with surface and then the surface name.

max_surface *int* (obsolete statement)

As of version 2.19, this statement is optional. If used, it specifies the maximum number of surfaces that will be defined. Each surface may have many panels, including disjoint panels.

start_surface [*name*]

Start of surface definition block. The surface name may be given with *name*, or it may be given afterwards with the name statement. If the name has not been used yet for a surface, then a new surface is started. Between this instruction and "end_surface", all lines need to pertain to surfaces. Parameters of one surface can be listed in multiple blocks, or parameters for many surfaces can be listed in one block.

new_surface *name*

Defines a new surface called *name*, but does not start a surface block. This statement is largely redundant with start_surface.

* name *name*

Name of the surface for editing. This statement is not required because the surface name can also be given with `start_surface`. This statement gives the name of the current surface for editing, and creates a new surface if needed.

* `action` *species(state) face action*
old format (version 2.18 and before): * `action` *face species action*
The behavior of molecules named *species* (and in state *state*, which is assumed to be solution if it's not entered) when they collide with the *face* face of this surface. *face* can be "front", "back", or "both". If *species* is "all", then this action applies to all molecules. The action can be "reflect", "absorb", "transmit", "jump", "port", or "periodic." In addition, it's permissible to enter "multiple," in which case the rates need to be set with `rate`; alternatively, just setting the rates will automatically set the action to "multiple." The default is transmission for all molecules.

* `rate` *species(state) state1 state2 value* [*new_spec*]
(Obsolete form: `rate` *face molec action value*)
The rate constant for transitions from *state1* to *state2* of molecules named *species* at this surface. For the species name, in *species*, "all" is permitted; however, "all" is not permitted anywhere else. Usually, *state* is omitted, but see below for where it is needed. *state1* and *state2* can be any of: fsoln, bsoln (in solution, hitting the front or back of the panel, respectively), front, back, up, or down. *value* is the rate constant or rate coefficient. If *new_spec*, which is an optional parameter, is entered, then molecules change to the listed species at the same time as changing states.

To specify interaction rates for molecules that collide with surface B, while diffusing along surface A, use the first *state* parameter. In this case: *state* is the starting surface-bound state on surface A; *state1* is fsoln to indicate collision with the front side of surface B or bsoln to indicate collision with the back side of surface B; and *state2* is fsoln or bsoln to indicate transmission through surface B and still bound to surface A (but cannot equal *state1*) or *state2* can be a surface-bound state to indicate that the molecule hops from surface A to surface-bound on surface B.

* `rate_internal` *species(state) state1 state2 value* [*new_spec*]
(Obsolete form: `rate_internal` *face molec action value*)
This is identical to rate, except that a slightly different value is entered. Instead of entering the surface action rate, enter the probability of the action at each collision. Probabilities for reflection are ignored since they are calculated as the probability that the molecule does not transmit, absorb, or jump.

*`neighbor_action` *action*
Behavior of surface-bound molecules when they collide with a panel that is a neighbor of the panel that they are bound to. There are only two

options: “hop” indicates that the molecule should hop onto the new panel with a 50% probability and stay with a 50% probability, and “stay” indicates that the molecule should stay on its own surface. The default is “stay”.

* `color` *face color* [*alpha*]
* `color` *face red green blue* [*alpha*]

Color of the *face* face of the surface. *face* can be “front”, “back”, or “both”. In the first format, *color* is a single word color, such as “red”, “green”, “magenta”, “cyan”, etc. In the second format, color values are numbers between 0 and 1, where 1 is maximum intensity a 0 is minimum (1 1 1 is white). The *alpha* value is optional and describes the opacity of the surface. If entered, it also needs to be between 0 and 1, where 1 is an opaque object (the default) and 0 is transparent. OpenGL graphics do not work well with non-integer alpha values, so don’t expect good results.

* `thickness` *float*

Boldness of the surface in pixels for drawing purposes. This is only relevant for 1-D and 2-D simulations, and for 3-D simulations in which surfaces are drawn with just vertices or edges and not faces.

* `stipple` *factor pattern*

Stippling of the surface edges, for drawing purposes. This is only relevant for 3-D simulations in which surfaces are drawn with just edges and not faces, and with opengl_good or better display method. In *factor*, which is an integer, enter the repeat distance for the entire stippling pattern (1 is a good choice). In *pattern*, which is a hexidecimal integer, enter the stippling pattern between 0x0000 and 0xFFFF. 0x00FF has long dashes, 0x0F0F has medium dashes, 0x5555 has dots, etc. Turn stippling off with 0xFFFF.

* `polygon` *face drawmode*

Drawing method for the *face* face of the surface. *face* can be “front”, “back”, or “both”. *drawmode* may be “none”, “vertex”, “edge”, “face”, or combinations of ‘v’, ‘e’, or ‘f’ for multiple renderings of vertices, edges, and/or faces. 2-D spheres and hemispheres are either filled or are outlined depending on the `polygon front` character.

* `shininess` *face value*

Shininess of the surface for drawing purposes. This value can range from 0 for visually flat surfaces to 128 for very shiny surfaces. This is only relevant for some simulations.

* `max_panels` *shape int* (obsolete statement)

Optional statement. This can be used to allocate memory for *int* panels of shape *shape* for this surface, although it is usually best to let Smoldyn allocate memory as needed. The shape may be “rect” for a rectangle, “tri”

for a triangle, “sph” for a sphere, “cyl” for a cylinder, “hemi” for a hemisphere, or “disk” for a disk. The surface can include panels with different shapes.

* `panel` *shape float … float [name]*

Defines a new panel for the surface, where the panel has shape *shape*. The shape may be “rect” for a rectangle, “tri” for a triangle, “sph” for a sphere, “cyl” for a cylinder, “hemi” for a hemisphere, or “disk” for a disk. Following the shape are numbers for the panel position, where these depend on the shape. At the end, it is possible to enter a string to name the panel, although this is optional (default names are the numbers 0, 1, 2, …; names are used for jump surfaces). If the name was used before, then this does not create a new panel, but modifies the existing panel.

For “rect”, enter the axis number that the rectangle is perpendicular to, preceded by a ‘+’ if the panel front faces the positive axis and a ‘-’ if it faces the negative axis (these signs must be entered); then enter the coordinates of a corner point; then enter the dimensions of the rectangle in sequential order of the axes, omitting the one that it is perpendicular to. These dimensions are better called displacements because they are added to the corner that is entered, so they may be positive or negative. For example, for a square in a 3-D system that is perpendicular to the *y*-axis, has sides of length 10 and is centered about the origin, enter: “`panel rect +1 -5 0 -5 10 10`”. This same square could be entered as “`panel rect +1 5 0 5 -10 -10`” , or with other descriptions. A rectangle is always perpendicular to an axis.

For “tri”, enter the coordinates of the corners of the triangle. This is one number for 1-D; 4 for 2-D, and 9 for 3-D. For 1-D, the front of the triangle always faces the positive axis; rectangles are completely equivalent and more versatile. For 2-D, triangles are really lines and the front side of the line is the side on the right when traveling in sequential order of the points that are entered. For 3-D, the triangle front is determined by the winding direction of the corners: if one is facing the front, the points wind counterclockwise. Unlike rectangles, triangles do not have to be perpendicular to axes.

For “sph”, enter the coordinates of the sphere center followed by the sphere radius and some drawing information. For 1-D, the center coordinate is a single number and the radius is entered next. For 2-D, the center coordinates are 2 numbers and then enter the radius followed by the number of sides on the polygon that should be drawn to represent the circle. For 3-D, the center coordinates are 3 numbers and then enter the radius, followed by the number of slices (longitude lines) and stacks (latitude lines) that are used for drawing the sphere. In the 2-D and 3-D cases, the drawing entries are used only for drawing; the circle or sphere

functions as an accurate smooth shape. For all dimensions, enter a positive radius to have the front of the surface on the outside and a negative radius for it to be on the inside.

For "cyl", enter the coordinates of the cylinder-axis start point and the cylinder-axis end point, then the radius, and then drawing information if appropriate. Cylinders are not permitted in 1-D. In 2-D, two numbers give the start point and two give the end point, followed by the radius. No drawing information is needed. In 3-D, enter three numbers for the start point, three for the end point, the radius, and then the number of slices and the number of stacks. For all dimensions, enter a positive radius to have the front of the surface on the outside and a negative radius for it to be on the inside.

For "hemi", enter the coordinates of the hemisphere center, the radius, and then the vector that points straight out of the hemisphere. Hemispheres are not permitted in 1-D. In 2-D, the center coordinates are 2 numbers, the radius is 1 number, the outward vector is 2 numbers, and finally enter the number of slices. For 3-D, the center is 3 numbers, the radius is 1 number, the outward vector is 3 numbers, and then enter 2 numbers for the numbers of slices and stacks. The outward pointing vector does not need to be normalized to unit length. For all dimensions, enter a positive radius to have the front of the surface on the outside and a negative radius for it to be on the inside.

For "disk", enter the coordinates of the disk center, the radius, a vector that points away from the front of the disk, and drawing information if appropriate. Disks are not permitted in 1-D. In 2-D, the center coordinates are 2 numbers, the radius is 1 number, and the normal vector is 2 numbers. For 3-D, the center coordinates are 3 numbers, the radius is 1 number, the normal vector is 3 numbers, and the number of drawing slices is entered last. Normal vectors do not need to have unit length.

* `jump` *name face -> name2 face2*
* `jump` *name face <-> name2 face2*

Defines a molecule jumping condition for a face of a single panel. This panel has name given with *name*, and face *face*. The name of a panel can be defined with the panel statement, or the default is just the shape and panel number (rect0, sph5, etc.). A molecule that hits this face of the panel, and that has "jump" action for this face, gets translated to the face *face2* of the panel named *name2* (which needs to be the same shape as the originating panel). A unidirectional arrow implies just jumping from the first panel to the second, whereas a double-headed arrow implies jumping in both directions.

* `neighbors` *panel neigh1 neigh2 …*

Defines a list of panels that neighbor the panel named *panel*. Surface-bound molecules can only diffuse from a panel to its neighbor if the neighbors are defined in this way. This statement is unidirectional in that it only sets, for example, *neigh1* as a neighbor of panel *panel* but not vice versa. If the neighboring panel is not part of the same surface as the origin panel, then specify the neighboring panel using *surface*:*panel*.

* `unbounded_emitter` *face species amount* pos_0 pos_1 … pos_{dim-1}

Declares a molecular source for which this surface should absorb molecules so as to yield a concentration distribution that is the same as that which would arise with unbounded diffusion. This statement does not create the molecular source, but only sets the panel absorption coefficients to yield the correct concentrations, assuming the emitter is created elsewhere (such as with a command or a zeroth order reaction). *face* is the side of the surface that faces the emitter, *species* is the emitted molecular species, *amount* is the emission rate (it only matters if there is more than one emitter for this surface and species, and then it is only the relative rates of the different emitters that matters), and *pos* is the system-dimensional position of the emitter. This statement is designed to be used with all emitters strictly inside a closed surface and all of them with positive amount values; however, neither of these criteria are checked, so other options can be used although no promises are made regarding their behaviors.

* `end_surface`

End of a block of surface definitions. Surface statements are no longer recognized but other simulation statements are.

`epsilon` *float*

See “simulation settings” section. This is not entered in a surface block.

`margin` *float*

See “simulation settings” section. This is not entered in a surface block.

`neighbor_dist` *float*

See “simulation settings” section. This is not entered in a surface block.

Statements about compartments

The statements shown below that are preceded by an asterisk need to be entered within compartment blocks, which start with `start_compartment` and end with `end_compartment`. Most of these statements can also be entered directly, preceded by the statement `compartment` and then the compartment name. Both forms are shown below.

`start_compartment` *name*

Start of compartment definition block. The compartment name may be given with *name*, or it may be given afterwards with the `name` statement. If the name has not been used yet for a compartment, then a new compartment is started. Between this instruction and "`end_compartment`", all lines need to pertain to compartments. Parameters of one compartment can be listed in multiple blocks, or parameters for many compartments can be listed in one block.

`new_compartment` *name*

Defines a new compartment called *name*, but does not start a compartment block. This statement is largely redundant with `start_compartment`.

* `name` *name*

Name of the compartment for editing. This statement is not required because the compartment name can also be given with `start_compartment`. This statement gives the name of the current compartment for editing, and creates a new compartment if needed.

* `surface` *surface*
`compartment` *name* `surface` *surface*

Name of a bounding surface for this compartment.

* `point` $pos_0 \ldots pos_{dim-1}$
`compartment` *name* `point` $pos_0 \ldots pos_{dim-1}$

An interior-defining point for this compartment.

* `compartment` *logic compartment*
`compartment` *name* `compartment` *logic compartment*

Logically combines the compartment being defined as it has been defined so far with the compartment that is listed in this statement. The logic options are: `equal`, `equalnot`, `and`, `andnot`, `or`, `ornot`, and `xor`.

* `end_compartment`

End of a block of compartment definitions. Compartment statements are no longer recognized but other simulation statements are.

Statements about reactions

`reaction` *rname* $reactant_1$ + $reactant_2$ -> $product_1$ + $product_2$
`reaction` *rname* $reactant_1$ + $reactant_2$ -> $product_1$ + $product_2$ *rate*

This defines a new reaction which is named *rname*, has a list of reactants, a list of products, and rate equal to *rate*. If there are no reactants, meaning that it is zeroth order, enter '0' as the reactant. Similarly, if there are no products, enter '0' as the sole product. The rate value is optional. As usual, enter species states in parentheses after the species names; "all" is permitted for reactant states, but not for product states.

`reaction_cmpt` *cname rname reactant*$_1$ + *reactant*$_2$ -> *product*$_1$ + *product*$_2$
`reaction_cmpt` *cname rname reactant*$_1$ + *reactant*$_2$ -> *product*$_1$ + *product*$_2$ *rate*
This is identical to the `reaction` statement except that this reaction only occurs within the compartment *cname*. All reaction orders work. For bimolecular reactions, both reactants need to be in the compartment for the reaction to occur.

`reaction_surface` *sname rname reactant*$_1$ + *reactant*$_2$ -> *product*$_1$ + *product*$_2$
`reaction_surface` *sname rname reactant*$_1$ + *reactant*$_2$ -> *product*$_1$ + *product*$_2$ *rate*
This is identical to the `reaction` statement except that this reaction only occurs on the surface *sname*. All reaction orders work. For bimolecular reactions, at least one reactant needs to be bound to the named surface for the reaction to occur.

`reaction_rate` *rname rate*
Sets the rate constant to *rate* for reaction named *rname*.

`confspread_radius` *rname rad*
Defines reaction *rname* as a conformational spread reaction. This reaction must have two reactants and up to two products. If it has two products, which is the most common case, then the first reactant gets replaced by the first product, and the second with the second. They keep their serial numbers and locations. The reaction domain extends over the radius that is listed here (this is effectively a binding radius). If this is entered, the reaction rate constant is interpreted as a first order rate constant.

`binding_radius` *rname rad*
Sets the binding radius of reaction *rname* to *rad*.

`reaction_probability` *rname prob*
A fixed probability value for unimolecular or bimolecular reactions. For unimolecular reactions, this is the probability of a reaction during one time step. For bimolecular reactions, this is the probability of a reaction occurring, given that the reactants are already closer than their binding radius. Here, the default value is 1, which is assumed in all rate calculations. For conformational spread reactions, this value can be used to directly enter the reaction probability at each time step, rather than letting it be calculated from the rate value. For regular bimolecular reactions, this can be used to adjust the effective reaction activation energy, although the theory has not been derived for that yet.

`reaction_production` *rname value*
Molecule production rate for zeroth order reactions. Instead of entering the reaction rate with `reaction_rate`, this allows on to enter the expectation number of molecules per time step in the entire simulation volume.

`product_placement` *rname type parameters*

Placement method and parameters for the products of reaction *rname*. This also affects the binding radius of the reverse reaction, as explained in the text. The type `irrev` requires no parameters. Types `pgem`, `pgemmax`, `pgemmaxw`, `ratio`, `unbindrad`, `pgem2`, `pgemmax2`, and `ratio2` each require one parameter. Types `offset` and `fixed` each require first a product molecule name and then a *dim*-dimensional vector as the parameter list. If multiple products are identical, then this placement instruction will only be applied to the first of the identical products. For this reason, you can also specify that this statement applies to the *n*'th product by entering the product name as `product_`*n* (e.g. `product_2` for the second product). The default placement method for reversible reactions is `pgemmaxw` (the terminal 'w' implies that a warning will be issued) with a parameter of 0.2. While it is suggested that placement types be entered with full words, single letter codes work as well. To create a "bounce" type reaction, for simulating excluded volume, enter the *type* as `bounce`. In this case, enter one parameter. For most purposes the best parameter is -1, which indicates that the reactant edges get separated by the same amount as they used to overlap (e.g. consider two reactants each of radius 1, so the binding radius is set to 2; then, if the center-to-center distance is found to be 1.6, the molecules get separated to make the center-to-center distance equal to 2.4). Alternatively, you can use the parameter value to define the new separation, which should be larger than the binding radius.

`reaction_serialnum` *rname rule_list*

Define rules for product molecule serial number assignments during reaction *rname*. There should be as many rule values as there are products for this reaction. For each product, choose from the following options: "`new`" for a new serial number (the default), "`r1`" for the serial number of the first reactant, "`r2`" for serial number of the second reactant (for second order reactions), "`p1`" for the serial number of the first product, "`p2`" for the serial number of the second product, etc., and an integer greater than zero for that value as the serial number. The codes can be separated by '+' symbols, as in the reaction definition, but this isn't required. Some of these options can lead to multiple molecules having the same serial numbers. This is allowed, but may lead to unexpected behavior in some runtime commands.

`reaction_intersurface` *rname rule_list*

Define rules to allow bimolecular reaction named *rname* to operate when its reactants are on different surfaces. In general, there should be as many rule values as there are products for this reaction. For each product choose "`r1`" if it should be placed on the first reactant's surface or relative to that surface, and "`r2`" if it should be placed on the second reactant's surface or relative to that surface (the relative conditions are for "soln" or "bsoln"

state products). The codes can be separated by '+' symbols, as in the reaction definition, but this isn't required. To turn off intersurface reactions, which is the default behavior, give *rule_list* as "off". To turn on intersurface reactions for reactions that have no products, give *rule_list* as "on".

`reaction_log` *filename rxnname serial_numbers*

Turns on reaction logging for all occurrences of the reaction *rxnname* and for molecules with serial numbers that are in the serial number list. The logging is sent to the file called *filename*. If the file is not `stdout`, then it should be declared with the `output_files` statement. Enter *rxnname* as "all" if all reactions should be logged. Likewise, enter the serial number list as "all" if reactions with all molecules should be logged. In the logging file, the output will be a single line of text for each occurrence of the reaction with the following items: the current simulation time, the name of the reaction, the location of the reaction (2 numbers for 2D, 3 for 3D), the serial numbers of each reactant, and the serial numbers of each product. If you request logging for a specific serial number, then an entry will be created if a molecule with this serial number is either a reactant or a product of the reaction (however, if it is the second or higher product, then the log entry will be missing the prior product serial numbers because these are not recorded as they are generated).

`reaction_log_off` *rxnname serial_numbers*

Turns off reaction logging for the reaction *rxnname* and for molecules with serial numbers that are listed in the serial number list. Either or both of *rxnname* and the serial number list can be "all".

Statements about ports

The statements shown below that are preceded by an asterisk need to be entered within port blocks, which start with `start_port` and end with `end_port`. Most of these statements can also be entered directly, preceded by the statement `port` and then the port name. Both forms are shown below.

`start_port` *name*

Start of port definition block. The port name may be given with *name*, or it may be given afterwards with the `name` statement. If the name has not been used yet for a port, then a new port is started. Between this instruction and "`end_port`", all lines need to pertain to ports. Parameters of one port can be listed in multiple blocks, or parameters for many ports can be listed in one block.

`new_port` *name*

Defines a new port called *name*, but does not start a port block. This statement is largely redundant with `start_port`.

* name *name*
Name of the port for editing. This statement is not required because the port name can also be given with start_port. This statement gives the name of the current port for editing, and creates a new port if needed.

* surface *surface*
port *name* surface *surface*
Name of the porting surface for this port.

* face *face*
port *name* face *face*
Face of the surface that is active for porting. Enter "front" or "back" for *face*.

* end_port
End of a block of port definitions. Port statements are no longer recognized but other simulation statements are.

max_port int (optional)
Maximum number of ports that may be defined.

Statements for lattices

The statements shown below that are preceded by an asterisk need to be entered within lattice blocks, which start with start_lattice and end with end_lattice. Most of these statements can also be entered directly, preceded by the statement lattice and then the lattice name. Both forms are shown below.

start_lattice *name*
Start of the lattice block. The lattice name may be given with *name*, or it may be given afterwards with the name statement. If the name has not been used yet for a lattice, then a new lattice is started. Between this statement and end_lattice, all lines need to pertain to lattices. Parameters of one lattice can be listed in multiple blocks, or parameters for many lattices can be listed in one block.

* name *name*
Name of the lattice for editing. This statement is not required because the lattice name can also be given with start_lattice. This statement gives the name of the current lattice for editing, and creates a new lattice if needed.

* type *type*
Type of the lattice. At present, this accepts two *type* strings, "nsv" and "pde", which stand for next-subvolume method and partial differential

equation method, respectively. However, only the NSV method has been implemented, so that's the only type that should be entered. This statement is optional, with NSV assumed if it is not entered.

* `port` *port*

Name of the port that the lattice uses to exchange molecules with the particle-based simulation.

* `boundaries` *dim pos1 pos2*
* `boundaries` *dim pos1 pos2 type*

Creates lower and upper boundaries for the lattice region, where *dim* is the dimension that is bounded in this statement, *pos1* is the lower bound, and *pos2* is the upper bound. In the second form, *type* is a character that represents the boundary type, which may be 'r' for reflective or 'p' for periodic. This syntax is essentially identical to the `boundaries` statement that is used to define the particle-based simulation volume.

* `lengthscale` *x1*
* `lengthscale` *x1 x2*
* `lengthscale` *x1 x2 x3*

Specifies the partition spacing within the lattice region of space. Use the first form for 1D systems, the second for 2D systems, and the third for 3D systems. The partition spacing values should be even divisors of the lattice dimensions that are given with the boundaries statement.

* `species` *species1 species2 ...*

List of species that should be used in the lattice region of space. These species need to have been declared previously in the particle region of space. This line may be entered multiple times. Rather than listing all species, the "all" keyword can be used to state that all of the current particle-side species should also be used on the lattice side.

* `reaction` *reaction1 reaction2 ...*
* `reaction move` *reaction1 reaction2 ...*

List of reactions that should be used in the lattice region of space. These reactions need to have been fully defined previously in the particle region of space. Rather than listing all reactions, the keyword "all" can be used to state that all of the current particle-side reactions should also be functional on the lattice side. If the keyword "move" is given in the list, as in the latter form above, then all subsequent listed reactions are "moved" to the lattice side, meaning that they are functional on the lattice side but become non-functional on the particle side. In this case, they are still defined on the particle side, but are simply disabled.

* `mol` *nmol name* pos_0 pos_1 … pos_{dim-1}

This adds molecules to the starting state of the simulation in the lattice region of space. This statement is essentially identical to the statement with the same name that is in the particle portion of the configuration file. The lattice regions starts with *nmol* type *name* molecules at location *pos*. Each of the *dim* elements of the position may be a number to give the actual position of the molecule or molecules; or the letter 'u' to indicate that the position for each molecule should be a random value between the bounding walls, chosen from a uniform density; or a position range which is given as two numbers separated with a hyphen.

* end_lattice

End of a block of lattice definitions. Lattice statements are no longer recognized but other simulation statements are.

Statements for simulation settings

`random_seed` *int*

Seed for random number generator. If this line is not entered, the current time is used as a seed, producing different sequences for each run. (This statement was called `rand_seed` through version 2.28.)

`accuracy` *float*

A parameter that determines the quantitative accuracy of the simulation, on a scale from 0 to 10. Low values are less accurate but run faster. Default value is 10, for maximum accuracy. Bimolecular reactions are only checked for pairs of reactants that are both within the same virtual box when accuracy is 0 to 2.99, reactants in nearest neighboring boxes are considered as well when accuracy is 3 to 6.99, and reactants in all types of neighboring boxes are checked when accuracy is 7 to 10.

`molperbox` *float*

Virtual boxes are set up initially so the average number of molecules per box is no more than this value. The default value is 5. `boxsize` is an alternate way of entering comparable information.

`boxsize` *float*

Rather than using `molperbox` to specify the sizes of the virtual boxes, `boxsize` can be used to request the width of the boxes. The actual box volumes will be no larger than the volume calculated from the width given here.

`gauss_table_size` *int*

This sets the size of a lookup table that is used to generate Gaussian-distributed random numbers. It needs to be an integer power of 2. The

default value is 4096, which should be appropriate for nearly all applications.

`epsilon` *float*

Maximum allowed distance separation between a surface-bound molecule and the surface. The default value, which is extremely small, is good for most applications.

`margin` *float*

The distance inside of a panel edge to which Smoldyn moves surface-bound molecules that diffuse off of a panel. The default value, which is extremely small, is good for most applications.

`neighbor_dist` *float*

Maximum distance that surface-bound molecules will jump across space to diffuse from one panel to a neighboring panel. In Smoldyn 2.37 and higher versions, the default for this value is extremely small, just large enough to prevent round-off error. It should not need to be changed. In prior versions, the default value was 3 times the maximum rms step length of surface-bound molecules, which was necessary due to a different surface-bound molecule diffusion algorithm.

4.2 Deprecated statements

The following statements are supported for backward compatibility and might be removed in future Smoldyn releases.

`max_species` *int*

This tells Smoldyn to allocate this many molecule species. As of version 2.23, new species are allocated as needed.

`max_names` *int*

Maximum number of molecular species that will be used. Statement has been superseded by `max_species`*, which is now also obsolete.*

`name` *name*

Name of a molecule. Statement has been superseded by `species`*.*

`names` $name_1$ $name_2$ … $name_n$

Names of all of the types of molecules present in the system. Statement has been superseded by `species`*.*

`max_compartment` *int*

Maximum number of compartments that will be defined. New compartments are now allocated as needed.

`reaction_permit` *rname state*
`reaction_permit` *rname* $state_1$ $state_2$

Allow reaction rname to happen for the specified reactant states, which need to be listed in the same order in which the reactants were entered. This function has not proven to be useful; instead, it's best to enter each reaction individually, rather than to have some reactions that operate on multiple states.

`reaction_forbid` *rname state*
`reaction_forbid` *rname* $state_1$ $state_2$

Forbid reaction rname to happen for the specified reactant states, which need to be listed in the same order in which the reactants were entered. Again, this function has not proven to be useful.

`pthreads` *int*

Sets the number of threads that Smoldyn should run with. Enter 0 for unthreaded operation (the default) or a larger number (including 1) for multi-threaded operation. None of the multi-threaded functions have been tested thoroughly, so they may create incorrect results or program crashes. They are also not substantially faster than the unthreaded versions of the same functions. Thus, use of multi-threaded operation is not generally recommended.

The following reaction statements are for block-style input and are only supported for backward compatibility.

`start_reaction`
`start_reaction` *order*

Start of reaction definition. Between this instruction and "`end_reaction`*", all lines need to pertain to this order of reaction. It is permissible to list reactions of the same order in multiple blocks, provided that only the first block includes a* `max_rxn` *statement and that sufficient reactions are declared with that statement.*

`order` *int*

Order of the reactions being declared (0, 1, or 2).

`max_rxn` *max_rxn*

Maximum number of reactions that will be declared of the given order.

`reactant` $r_0\ r_1 \ldots r_{nrxn-1}$
`reactant` *name(state)* $r_0\ r_1 \ldots r_{nrxn-1}$
`reactant` $name_1(state_1) + name_2(state_2)\ r_0\ r_1 \ldots r_{nrxn-1}$

Declaration of reactants and reaction names for zeroth order, unimolecular, and bimolecular reactions, respectively. The listed molecule names are the reactants and the following strings are the respective reaction names. Note that there are spaces before and after the '+' symbol. States are optional. If they are omitted, only the solution states are allowed to react, whereas if they are included then only the states listed can react. States may be set to "all".

`permit` *name(state) r value*
`permit` *name(state) + name(state) r value*

Set permission of reaction r, for the specified reactants, to value, which should be 0 or 1; 0 means the reaction does not happen, 1 means it does. The default depends on the states that were entered with the reactant statement, if any.

`rate` *r rate*

Reaction rate constant for reaction called r. Units for the reaction rate constant are $(volume)^{order-1}$ *times inverse time. These rates are converted by the program into probabilities or binding radii. To enter the simulation parameters directly, use* `rate_internal`*. Conformational spread reactions are entered with order 2, but this rate constant has units of inverse time.*

`confspread_radius` *r float*

Defines reaction r a a conformational spread reaction. This reaction must have two reactants and two products; upon reaction, the first reactant is

replaced by the first product, and the second with the second. The reaction domain extends over the radius that is listed here (this is effectively a binding radius).

`rate_internal` *r float*

Internal value for reaction rate information, which can be used to override the internal rates that are calculated from the `rate` *entry. For zeroth order reactions, this is the expectation total number of reactions per time step; for unimolecular reactions, this is the reaction probability per time step for each reactant molecule; and for bimolecular reactions, this is the binding radius. This has no effect for conformational spread reactions.*

`probability` *r float*

A fixed probability value for bimolecular reactions. The default value is 1, which is assumed in all rate calculations. However, this statement allows the user to set a different probability, which is ignored in all rate calculations, but can be useful in some situations. For conformational spread reactions, this value can be used to directly enter the reaction probability at each time step, rather than letting it be calculated from the rate value. For simple reactions, this can be used to adjust the effective reaction activation energy, although the theory has not been derived for that yet.

`product` *r name + name + … + name*
`product` *r name(state) + name(state) + … + name(state)*

List of products for reaction `r`*. States for products may be entered, which can be "solution", "fsoln", "bsoln", "front", "back", "up", or "down"; if no state is given, solution is assumed.*

`product_param` *r* `i`
`product_param` *r* `p,x,X,r,b,q,y,s` *float*
`product_param` *r* `o,f` *prod_name* pos_0 $pos_1 \ldots pos_{dim-1}$

Parameters for the initial placement of products of reaction `r`*. A product parameter also affects the binding radius of the reverse reaction. These are explained in section 3. In the first format, a type of '*`i`*' indicates that the reverse reaction is ignored for calculations. The second format uses one of the type letters shown: '*`p`*' and '*`q`*' are geminate rebinding probabilities, '*`x`*' and '*`y`*' are maximum geminate rebinding probabilities, '*`r`*' and '*`s`*' are ratios of unbinding to binding radii, and '*`b`*' is a fixed unbinding radius. The third format yields products that have a fixed relative orientation, which is either randomly rotated with '*`o`*', or not rotated with '*`f`*'. In the absence of better information, a useful default parameter type is either '*`x`*' or '*`y`*', with a value of about 0.2. '*`X`*' is identical to '*`x`*', and is intended only as an internal code to indicate that the user didn't enter a product_param line, which will yield a default value of 0.2 and a warning.*

end_reaction

End of reaction definition. Reaction instructions are no longer recognized but other simulation instructions are.

4.2 Runtime commands

Simulation control commands

`stop`
: Stop the simulation.

`pause`
: This puts the simulation in pause mode. If opengl graphics are used, continuation occurs when the user presses the spacebar. When graphics are not used, the user is told to press enter.

`beep`
: The computer beeps when this is reached. Nothing else is done.

`keypress` *char*
: Send a signal to the graphics manipulation component of the program to execute the behavior that would occur when a key is pressed. For the arrows, and shift-arrows, the character should be r for right, l for left, u for up, d for down, and the respective upper case characters for the shift-arrows.

`setflag` *number*
: Sets the global command flag value to *number*, which can be a floating point value. This is generally used after a conditional command, and is then queried by one or more `ifflag` commands.

`setrandseed` *seed*
: Sets the random number seed to the specified integer value. If the seed listed is less than 0, the current time is used for the seed.

`setgraphics` *type*
: Sets the display graphics to type *type*. If graphics were not set up initially, using the `graphics` statement, this command does nothing. Otherwise, options for type are "`opengl`" or "`opengl_good`".

`setgraphic_iter` *timesteps*
: Sets the graphics update interval to *timesteps* time steps. This is only functional if graphics were set up initially, using the `graphics` statement.

`updategraphics`
: Update the graphics window.

File manipulation commands

`overwrite` *filename*

Erases the output file called *filename* but leaves it open for more writing. (In versions 1.77 and earlier, this called another command.)

`incrementfile` *filename*
: A new output file is created based upon the *filename*. The first time this is called the filename is appended with a "`_001`", which is then incremented with subsequent calls to "`_002`", and so on. These numbers precede any suffix on the filename. (In versions 1.77 and earlier, this called another command.)

Conditional commands

`ifflag` *symbol number command*
: Run command *command*, depending on the value of the global command flag. Enter *symbol* as '<' if *command* should be run if the flag value is less than *number*, as '>' for the flag value greater than *number*, and as '=' for the flag value equal to *number*.

`ifprob` *value command*
: Run command *command* with probability *value*, which should be between 0 and 1. If you want to run multiple commands with this probability use the `setflag` and `ifflag` commands.

`ifno` *species*(*state*) *command*
: Run command *command* if no molecule of type *species* remains. The molecule state *state* is optional, with "solution" as a default. The name and/or the state may be "all".

`ifless` *species*(*state*) *num command*
: Run command *command* if there are fewer than *num* molecules of type *species* remaining. The molecule state *state* is optional, with "solution" as a default. The name and/or the state may be "all".

`ifmore` *species*(*state*) *num command*
: Run command *command* if there are more than *num* molecules of type *species*. The molecule state *state* is optional, with "solution" as a default. The name and/or the state may be "all".

`ifincmpt` *species*(*state*) *char number compartment command*
: Run command *command* depending on how the number of molecules of type *species* within compartment *compartment* compares with *number*. Enter *char* as '<' if *command* should be run with less than *number* molecules, as '>' for more than *number*, or as '=' for equal to *number*. A space is required between this symbol and *number*. The molecules' state *state* is optional, with "solution" as a default. The species and/or state may be "all".

`ifchange` *species*(*state*) *char num command*

Run command *command* if the number of molecules of type *species* changes (or doesn't change) relative to the previous time this command was called. The molecule state *state* is optional, with "solution" as a default. The name and/or the state may be "all". Enter *char* as '>' to respond to a difference that is greater than *num*, '<' to respond to a difference that is less than *num*, '=' to respond to a difference that is exactly equal to *num*, or '!' to respond to a difference that does not equal *num*. For example, enter *char* and *num* as "! 0" to respond to any change in molecule numbers, or as "< -10" to respond to any decrease that exceeds 10 molecules.

System observation commands

For all of the observation commands, if *filename* is the last parameter, then it may be omitted and results will be output to stdout.

`warnescapee` *species*(*state*) *filename*

Looks for molecules of type *species* that are outside the system boundaries, printing results to *filename*. If there are none, nothing is printed. For each molecule that escaped during the last time step, this prints a line of text that says where the molecule was, where it is, and where it crossed the last surface. Setting *species* to "all" allows all molecule types to be checked. This needs to be run at every time step to catch all escaping molecules.

`echo` *filename* "*string*"

Just prints the text within double quotes to the file, without any terminal newline. If you want a newline, include \n at the end of the string. The quotes need to be plain quotes, not curly quotes.

`molcountheader` *filename*

This prints one line of display with the word "time" and then the name of each molecule species. This is intended to be used as a header line for the `molcount`, `molcountinbox`, etc. commands.

`molcount` *filename*

Each time this command is executed, one line of display is printed to the listed file, giving the time and the number of molecules for each molecular species. Molecule states are ignored. The ordering used is the same as was given in the `species` command. This command accounts for molecules in lattices and their associated ports.

`molcountinbox` *xlow xhigh filename*
`molcountinbox` *xlow xhigh ylow yhigh filename*

`molcountinbox` *xlow xhigh ylow yhigh zlow zhigh filename*

Each time this command is executed, one line of display is printed to the listed file, giving the time and the number of molecules that are within the box that is defined by the low and high parameter, for each molecular species. Molecule states are ignored. The ordering used is the same as was given in the `species` statement.

`molcountincmpt` *compartment filename*

Each time this command is executed, one line of display is printed to the listed file, giving the time and the number of molecules that are within the compartment *compartment* for each molecular species. Only solution-phase molecules are listed. The ordering used is the same as was given in the `species` statement.

`molcountincmpts` $compartment_1$ $compartment_1$ … $compartment_n$ *filename*

Each time this command is executed, one line of display is printed to the listed file, giving the time and the number of molecules that are within each of the compartments listed, for each molecular species. Up to 16 compartments may be listed. Only solution-phase molecules are reported to the output. The molecule ordering used is the same as was given in the `species` statement.

`molcountincmpt2` *compartment state filename*

Identical to `molcountincmpt` except that this counts molecules that are in state *state*. Entering state as "all" means that molecules of all states are counted. Note that the surfaces that bound a compartment are included in that compartment.

`molcountonsurf` *surface filename*

Each time this command is executed, one line of display is printed to the listed file, giving the time and the number of molecules that are bound to the surface *surface* for each molecular species. The molecule state is not printed. The ordering used is the same as was given in the `species` statement.

`molcountspace` *species*(*state*) *axis low high bins average filename*
`molcountspace` *species*(*state*) *axis low high bins low high average filename*
`molcountspace` *species*(*state*) *axis low high bins low high low high average filename*

This command measures a line profile of molecules. It only counts molecules of type *species*, with an optional *state* specification, although *species* and/or *state* can be "all". The line profile is along axis number *axis*, which is 'x', 'y', or 'z' (or a number between 0 and the system dimensionality minus 1), extends from *low* to *high*, and is comprised of *bins* equally spaced bins (*i.e.* it's a histogram). These bins extend exactly from *low* to *high*, and thus do not count any molecules that are outside this range. For two dimensions, the line width and lateral position are

specified with another pair of *low* and *high* values; for three dimensions, two more pairs of *low* and *high* values are required which now specify the sides of a rectangular cross-section tube. The volume investigated includes all edges. To illustrate the sequence of parameters, suppose the command is used in a 3-D system to show concentration variation along the *y*-axis. In this case, axis is 1, the first *low* and *high* indicate the ends of the measurement range along the *y*-axis, the next *low* and *high* indicate the domain on the *x*-direction, and the third *low* and *high* indicate the domain on the *z*-direction. Set the average input to 0 to not use averaging, in which case there is output at every command execution. Otherwise, this only produces an output every average iterations, at which point it outputs means that were collected over the preceding iterations. At each output time, the command outputs a single line of text to *filename* with the time followed by the numbers (or average numbers) of the specified molecules in each histogram bin. This command accounts for molecules in lattices, but not for molecules in ports.

`molcountspecies` *species*(*state*) *filename*
: Prints out a single line of text to *filename* with time and the number of molecules of the listed species that are in state *state*. Either or both of *species* and *state* may be "all". If *state* is not included, solution is assumed.

`mollistsize` *listname filename*
: Prints out a single line of text to *filename* with the total number of molecules in the molecule list named *listname*. This is allowed to be "all".

`listmols` *filename*
: This prints out the identity, state, and location of every molecule in the system to the listed file name, using a separate line of text for each molecule.

`listmols2` *filename*
: This is very similar to `listmols` but has a slightly different output format. Each line of text is preceded by the "time counter", which is an integer that starts at 1 and is incremented each time the routine is called. Also, the species and state names of molecules are not printed, but instead the species and state numbers are printed.

`listmols3` *species*(*state*) *filename*
: This is identical to `listmols2` except that it only prints information about molecules of type *species*. *state* is optional; *species* and/or *state* can be "all".

`listmols4` *species*(*state*) *filename*

This is identical to `listmols3` except that it accounts for wrap-arounds when displaying molecule positions. In other words, if the *x*-axis ranges from 0 to 10, a molecule starts in the middle at $x = 5$ and diffuses to the right for a full lap, returning to the middle of the system, then its *x* value is printed here as 15, rather than as 5 as it is for `listmols3`. *state* is optional; *species* and/or *state* can be "all".

`listmolscmpt` *species*(*state*) *cmpt filename*

This prints out the time counter (see `listmols2`), species, state, and location of every molecule that is within compartment *cmpt*. It only prints information about molecules of type *species*. *state* is optional; *species* and/or *state* can be "all".

`molpos` *species*(*state*) *filename*

This prints out the time and then the positions of all molecules of type *species* on a single line of text, to the listed filename. *state* is optional; *species* and/or *state* can be "all".

`trackmol` *serno filename*

Outputs the time and the species, state, location, and inside vs. outside compartment status for each compartment of the single molecule with serial number *serno*. This stops after it finds the first molecule with the requested serial number.

`molmoments` *species*(*state*) *filename*

This prints out the positional moments of the molecule type given to the listed file name. All the moments are printed on a single line of text; they are the number of molecules, the mean position vector (*dim* values), and the variances on each axis and combination of axes (dim^2 values). *state* is optional; neither *species* nor *state* can be "all".

`savesim` *filename*

This writes the complete state of the current system to the listed file name, in a format that can be loaded in later as a configuration file. Note that minor file editing is often desirable before simulating a file saved in this manner. In particular, the saved file will declare its own name as an output file name, which will erase the configuration file.

`meansqrdisp` *species*(*state*) *dim filename*

This function is used to measure mean square displacements (diffusion rates) of molecules of type *species*, along dimension *dim* ('x', 'y', or 'z', or 0, 1, or 2) printing the results to *filename*. When it is first invoked, it records the positions of all molecules of type *species*. Then, and every subsequent time it is called, it compares the current positions of all molecules that still exist to the old ones, calculates the average squared displacement ($\langle r^2 \rangle$), and prints the time and that number to a single line in

the output file. If *dim* is "all", this sums the mean square displacement for all dimensions, otherwise *dim* should be a dimension number. As of version 1.73, this accounts for periodic boundaries. *state* is optional; neither *species* nor *state* can be "all". As of version 1.84, this prints out three numbers in each line: time, $\langle r^2 \rangle$, and $\langle r^4 \rangle$. This command does not work if multiple molecules have the same serial number (which can only happen if you use the `reaction_serialnum` statement).

`meansqrdisp2` *species*(*state*) *dim start report max_mol max_moment filename*
This function is an expanded version of `meansqrdisp`. As above, it measures mean square displacements of molecules of type *species*, along dimension *dim* ('x', 'y', or 'z', or 0, 1, or 2), and prints the results to *filename*. The *start* and *report* arguments control when this command starts tracking molecules and when it reports their mean square displacements, respectively. For *start*, enter 'i' to track molecules that exist when the command is initially invoked, enter 'c' to track those that are created after the first call, and enter 'a' (all) to track both sets of molecules. For *report*, enter 'e' to report on every molecule that is being tracked, or 'r' to report on only those that reacted since the command was last called. In this latter case, the position that is used for a reacted molecule is its most recently tracked position, since it no longer exists. For example, if you want to see how far molecules diffuse between their creation in one reaction and their destruction in another reaction, set *start* to 'c' and *report* to 'r'. Or, set *start* to 'i' and *report* to 'e' for this function to be identical to `meansqrdisp`. It can track up to *max_mol* molecules. This function prints out the time and then results for all moments, even and odd, from $\langle r^0 \rangle$ (the number of molecules being reported on) to $\langle r^{max\_moment} \rangle$. This command accounts for periodic boundaries. *state* is optional; neither *species* nor *state* can be "all". This command does not work if multiple molecules have the same serial number (which can only happen if you use the `reaction_serialnum` statement).

`meansqrdisp3` *species*(*state*) *dim start report max_mol change filename command*
This function is quite similar to `meansqrdisp` and `meansqrdisp2`. It measures mean square displacements of molecules of type *species*, along dimension *dim* ('x', 'y', or 'z', or 0, 1, or 2). Then, this function divides these values by the molecules' ages to compute effective diffusion coefficients and prints the results to *filename*. The effective diffusion coefficient average is weighted using the molecule ages, so that old molecules have proportionately greater weight in the average than young molecules. The *start* and *report* arguments control when this command starts tracking molecules and when it reports their mean square displacements, respectively. For *start*, enter 'i' to track molecules that exist when the command is initially invoked, enter 'c' to track those that are created after the first call, and enter 'a' (all) to track both sets of

molecules. For *report*, enter 'e' to report on every molecule that is being tracked, or 'r' to report on only those that reacted since the command was last called. In this latter case, the position that is used for a reacted molecule is its most recently tracked position, since it no longer exists. It can track up to *max_mol* molecules. This function prints out the time, the number of molecules reported on, and the effective diffusion coefficient. Note that the first command run will always output an effective diffusion coefficient of NaN (or #1.IND on Windows systems) due to a 0/0 division error arising from 0 displacement divided by 0 time difference. If the effective diffusion coefficient changed less than *change* since the last time this function was executed, then the command *command* is run (e.g. if *change* is 0.01 then a fractional diffusion coefficient change of 1% or less will cause *command* to be run). See the example file called meansqrdisp3.txt. This command accounts for periodic boundaries. The *species* cannot be "all". The state is optional. If the state is "all", then molecules for all states of this species must be stored in the same molecule list. No warning is issued if this is not the case, but some molecules simply won't be counted. This command does not work if multiple molecules have the same serial number (which can only happen if you use the `reaction_serialnum` statement).

`residencetime` *species*(*state*) *start report summary_out list_out max_mol filename*
This function computes residence times of individual molecules of type *species*, thus showing how long they have existed in the system. As with `meansqrdisp2` and `meansqrdisp3`, the *start* and *report* arguments control when this command starts tracking molecules and when it reports their residence times, respectively. For *start*, enter 'i' to track molecules that exist when the command is initially invoked, enter 'c' to track those that are created after the first call, and enter 'a' (all) to track both sets of molecules. For *report*, enter 'e' to report on every molecule that is being tracked, or 'r' to report on only those that reacted since the command was last called. It can track up to *max_mol* molecules. This function needs to be invoked at every time step so that molecules can be tracked accurately. However, output may not be wanted at every time step, so set *summary_out* to the number of invocations between when this should print a summary output, with the time number of molecules, and mean residence time, and set *list_out* to the number of invocations between when this should print a list output, with a list of all molecules with their IDs and their current ages. In both cases, set values to 0 or less to not have this type of output. *state* is optional; neither *species* nor *state* can be "all". See the residencetime.txt example file. This command does not work if multiple molecules have the same serial number (which can only happen if you use the `reaction_serialnum` statement).

`diagnostics` *type*

Displays diagnostics about various parts of the data structures to the screen. These are identical to those that are displayed at program initialization, but for later times. The options for the type word are: "simulation", "wall", "molecule", "surface", "command", "box", "reaction", "compartment", "port", "check", and "all".

`executiontime` *filename*

Prints a single line of text with the current simulation time and the number of seconds of real time that the simulation has taken to execute since the simulation was started.

`printLattice` *filename*

Displays diagnostics about all lattices.

`writeVTK` *filepath/filename*

Outputs VTK format data for viewing with applications such as Visit or Paraview. This creates a stack of files in the working directory, or somewhere else depending on the *filepath*, for which the names start with *filename*`Lattice00_00001.vtu` and *filename*`Molecules00001.vtu`, where *filename* is the entered file name. The *filepath* directory needs to have been created beforehand. In contrast to most filenames, this path and name should not be declared with the "`output_files`" statement. The filename numbers are incremented for each snapshot.

<u>System manipulation commands</u>

`set` *statement*

This command lets you use essentially any statement that can be entered in a configuration file. The statement can, for example, create new reactions, add surfaces, change rate constants, etc. It has not been fully debugged.

`pointsource` *species num* $pos_0\ pos_1 \ldots pos_{dim}$

Creates *num* new molecules of type *species* and at location *pos*. Molecule states are set to solution.

`volumesource` *species num* $pos_{0,\mathrm{low}}\ pos_{0,\mathrm{high}}\ pos_{1,\mathrm{low}}\ pos_{1,\mathrm{high}} \ldots pos_{dim,high}$

Creates *num* new molecules of type *species* and within the location bounded by pos_{low} and pos_{high}. Molecule states are set to solution.

`movesurfacemol` *species(state) prob* $surface_1{:}panel_1\ surface_2{:}panel_2\ [state_2]$

Moves molecules of type *species* and state *state*, and which are bound to $panel_1$ of $surface_1$, to $panel_2$ of $surface_2$ with probability *prob*. If entered, the new molecular state becomes $state_2$, which may be any state including fsoln or bsoln; otherwise the molecule state is unchanged. The new molecule location is at a random location on $panel_2$ (which contrasts the behavior of the jump statement). Either or both of $panel_1$ and $panel_2$ can

be “all” for all panels of the respective surface. Also, the *species* and/or the *state* may be “all”.

`killmol` *species*(*state*)
: Kills all molecules of type *species*. *state* is optional; *species* and/or *state* may be “all”.

`killmolprob` *species*(*state*) *probability*
: Kills some molecules of type *species*; the probability of a molecule being killed is *probability* (between 0 and 1). *state* is optional; *species* and/or *state* may be “all”.

`killmolinsphere` *species*(*state*) *surface*
: Kill all molecules of type *species* that are in any sphere that is a panel of surface *surface*. If *surface* is “all” then every surface is scanned. *state* is optional; *species* and/or *state* may be “all”.

`killmolincmpt` *species*(*state*) *compartment*
: Kill all molecules of type *species* that are in compartment *compartment*. *state* is optional; *species* and/or *state* may be “all”.

`killmoloutsidesystem` *species*(*state*)
: Kill all molecules of type *species* that are outside of the system boundaries. *state* is optional; *species* and/or *state* may be “all”.

`fixmolcount` *species num*
: Counts the number of solution-phase molecules of type *species*. If this count is more than *num*, then molecules of this type, chosen at random, are removed until *num* is reached. Otherwise, molecules are added to random positions in the system to achieve a total count of *num*. This function considers the entire system volume.

`fixmolcountrange` *species low_num high_num*
: Exactly like `fixmolcount`, except that the molecule count is unchanged if it is between *low_num* and *high_num*, and is otherwise modified to bring it to within the range.

`fixmolcountonsurf` *species*(*state*) *num surface*
: Counts the number of surface-bound molecules of type *species* and state *state*. If this count is more than *num*, then molecules of this type, chosen at random, are removed until *num* is reached. Otherwise, molecules with the proper state are added to random positions on the surface to achieve a total count of *num*.

`fixmolcountrangeonsurf` *species(state) low_num high_num surface*

Exactly like `fixmolcountonsurf`, except that the molecule count is unchanged if it is between *low_num* and *high_num*, and is otherwise modified to bring it to within the range.

`fixmolcountincmpt` *species num compartment*
Counts the number of solution-phase molecules of type *species* and in compartment *compartment*. If this count is more than *num*, then molecules of this type, chosen at random, are removed until *num* is reached. Otherwise, molecules are added to random positions in the compartment to achieve a total count of *num*.

`fixmolcountrangeincmpt` *species low_num high_num compartment*
Exactly like `fixmolcountincmpt`, except that the molecule count is unchanged if it is between *low_num* and *high_num*, and is otherwise modified to bring it to within the range.

`equilmol` $species_1(state_1)$ $species_2(state_2)$ *prob*
Equilibrate these molecules. All molecules of type $species_1$ and $species_2$ will be randomly replaced with one of the two types, where type $species_2$ has probability *prob*. $state_1$ and $state_2$ are optional; defaults are "solution". Neither species nor states may be "all".

`replacexyzmol` *species*(*state*) pos_0 pos_1 … pos_{dim-1}
If there is a non-diffusing molecule at exactly position *pos*, it is replaced with one of type *species*. This command stops after one molecule is found. *state* is optional and may not be "all"; default is solution.

`replacevolmol` $species_1(state_1)$ $species_2(state_2)$ *frac* $pos_{0,\text{low}}$ $pos_{0,\text{high}}$ $pos_{1,\text{low}}$ $pos_{1,\text{high}}$ … $pos_{dim-1,\text{high}}$
Fraction *frac* molecules of type $species_1$ in the volume bounded by pos_{low}, pos_{high} are replaced with ones of type $species_2$. States are optional and are solution by default; neither species nor states may be "all".

`replacecmptmol` $species_1(state_1)$ $species_2(state_2)$ *frac compartment*
Fraction *frac* molecules of type $species_1$ in the compartment named *compartment* are replaced with ones of type $species_2$. States are optional and are solution by default; neither species nor states may be "all".

`modulatemol` $species_1(state_1)$ $species_2(state_2)$ *freq shift*
Modulates molecules of types $species_1$ and $species_2$, just like `equilmol`, but with a variable probability. Every time this command executes, any of the two types of molecules in the system are replaced with a molecule of type $species_1$ with probability $\cos(freq*t+shift)$, where t is the simulation time, and otherwise with a molecule of type $species_2$. States are optional and are solution by default; neither species nor states may be "all".

`react1` *species*(*state*) *rxn*

All molecules of type *species* are instantly reacted, resulting in the products and product placements given by the unimolecular reaction named *rxn*. Note that *species* does not have to be the normal reactant for reaction *rxn*. The state is optional; *species* and/or *state* may be "all".

`setrateint` *rxn rate*

Sets the internal reaction rate of the reaction named *rxn* to *rate*. See the description above for `rate_internal` for the meanings of *rate* for the different reaction orders.

`shufflemollist` *listname*

Randomly shuffles the sequence of molecules in the molecule list called *listname*. Enter "`all`" for all lists. This is useful for systems that are especially ordered or for simulations with unusually long time steps, because these situations may make simulation results depend on the molecule list sequences.

`shufflereactions` *reactant1 reactant2*

Randomly shuffles the sequence of bimolecular reactions that reactant species `reactant1` and `reactant2` can undergo (all states are indexed together). Either or both or `reactant1` and `reactant2` can be "all".

Deleted, starting with Smoldyn 2.19:

`setsurfcoeff` surface_name species state1 state2 rate

Sets the rate coefficient for the conversion of species from state1 to state2 at surface surface_name.

`settimestep` *dt*

Changes the simulation time step to *dt*. This changes the diffusion step lengths, reaction binding and unbinding radii, and surface action probabilities. Caution should be used if the time step is increased to longer than the original value because no checks are made to ensure that the simulation will still yield accurate results.

`porttransport` *port1 port2*

Transports molecules from the output buffer of *port1* to the input of *port2*. These may be the same ports.

`excludebox` *xlo xhi*
`excludebox` *xlo xhi ylo yhi*
`excludebox` *xlo xhi ylo yhi zlo zhi*

This keeps all molecules from entering a rectanguloid box within the system volume. Use the first form for one dimension, the second for two dimensions, and the third for three dimensions. Molecules that start within the box can stay there, but any molecule that tries to diffuse into the

box is returned to its location at the previous time step. This command needs to be run at every time step to work properly.

`excludesphere` *x rad*
`excludesphere` *x y rad*
`excludesphere` *x y z rad*
This keeps all molecules from entering a sphere within the system volume. Use the first form for one dimension, the second for two dimensions, and the third for three dimensions; the coordinates given are the sphere center and *rad* is the sphere radius. Molecules that start within the sphere can stay there, but any molecule that tries to diffuse into the sphere is returned to its location at the previous time step. This command needs to be run at every time step to work properly.

`includeecoli`
An *E. coli* shape is defined as a cylinder with hemispherical endcaps, where the long axis of the bacterium extends the length of the *x*-axis within the system walls and the radius of both the cylinder and the endcaps is half the spacing between the walls that bound the *y*-axis. This command moves any molecule that diffuses out of the *E. coli* shape back to its location at the previous time step, or to the nearest surface of the *E. coli* if it was outside at the previous time step as well. This command does not need to be run at every time step to work properly. This only works with a 3 dimensional system.

`setreactionratemolcount` *rxn c0 c1 species1*(*state1*) *c2 species2*(*state2*) ...
This sets the rate of the reaction named *rxn* to: *c0*, plus *c1* times the number of molecules of *species1* and *state1*, plus *c2* times the number of molecules of *species2* and *state2*, plus any additional coefficients and species that are listed here. Species and/or states may be "all" and wildcards are permitted. If the reaction rate is computed to be a negative value, it is set to zero instead (and no warning is issued).

`expandsystem` *expandx expandy expandz*
Expand, or contract, everything in system, which includes molecule locations and surfaces, about the center of the system. Expands by *expandx* along the x-coordinate, by *expandy* along the y-coordinate, and by *expandz* along the z-coordinate. Enter as many numbers as there are dimensions. Each number should be 1 for no change, a number larger than 1 for expansion and a number smaller than 1 for contraction. Negative numbers perform system inversion. This command can be used, for example, to mimic lengthwise or diameter growth of a cell. *Warning*: isotropic expansion or contraction, in which all three expansion values are equal, generally works well, with no unintentional transfer of molecules across surfaces. However, anisotropic expansion or contraction would normally cause some of Smoldyn's panel shapes to become distorted,

including spheres, hemispheres, cylinders, and disks. Smoldyn does not support this, so these panels are expanded but not distorted. In the process, molecules often cross the surfaces unintentionally and need to be dealt with separately (e.g. killed off). See the expandsystem.txt example file.

5. Copyright and Citation

If you use Smoldyn to a significant extent, then I would appreciate citations to the research papers that describe the program, as appropriate. These papers are:

Description of the Smoldyn program
• Andrews, Steven S., Nathan J. Addy, Roger Brent, and Adam P. Arkin "Detailed simulations of cell biology with Smoldyn 2.1" *PLoS Comp. Biol.* 6:e1000705, 2010.

How to use Smoldyn
• Andrews, Steven S. "Smoldyn User's Manual" http://www.smoldyn.org, 2011.
• Andrews, Steven S. "Spatial and stochastic cellular modeling with the Smoldyn simulator" *Methods for Molecular Biology*, 804:519-542, 2012.

Description of Smoldyn algorithms
• Andrews, Steven S. and Dennis Bray "Stochastic simulation of chemical reactions with spatial resolution and single molecule detail" *Phys. Biol.* 1:137-151, 2004.
• Andrews, Steven S. "Accurate particle-based simulation of adsorption, desorption, and partial transmission" *Phys. Biol.* 6:046015, 2009.

Nearly all of the core Smoldyn program was written by myself (Steve Andrews), with the exceptions being a few short routines that were copied from *Numerical Recipes in C* (Press, Flannery, Teukolsky, and Vetterling, Cambridge University Press, 1988), which are acknowledged where appropriate. The compiled version of Smoldyn, the components of the source code that are not copyrighted by others, and this documentation are copyrighted by myself. It is distributed under the terms of the Lesser Gnu General Public License (LGPL). No warranty is made for the performance or suitability of any portion of Smoldyn.

I expect to maintain a working copy of the program indefinitely. The download site for Smoldyn is http://www.smoldyn.org, where the program may be obtained for free. If improvements are made to the code or bugs are fixed, then I would appreciate a copy of the modified source code. If you find any bugs in the code, please let me know! My e-mail address is steven.s.andrews@gmail.com.

Following are the text of the LGPL and the GPL.

GNU LESSER GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>. Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions. As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License. "The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below. An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library. A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version". The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version. The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL. You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions. If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version: a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files. The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following: a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works. You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following: a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. b) Accompany the Combined Work with a copy of the GNU GPL and this license document. c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. d) Do one of the following: 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries. You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following: a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License. The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation. If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

GNU GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>. Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble. The GNU General Public License is a free, copyleft license for software and other kinds of works. The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too. When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things. To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others. For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights. Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it. For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions. Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users. Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free. The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions. “This License” refers to version 3 of the GNU General Public License. “Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks. “The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations. To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work. A “covered work” means either the unmodified Program or a work based on the Program. To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well. To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying. An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code. The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work. A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language. The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it. The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work. The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source. The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions. All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law. You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you. Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law. No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures. When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising

rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. **Conveying Verbatim Copies.** You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program. You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. **Conveying Modified Source Versions.** You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions: (a) The work must carry prominent notices stating that you modified it, and giving a relevant date. (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to “keep intact all notices”. (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. (d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so. A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. **Conveying Non-Source Forms.** You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways: (a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. (b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. (c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. (d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. (e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d. A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work. A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product. “Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made. If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM). The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network. Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. **Additional Terms.** “Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions. When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission. Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms: (a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or (b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or (c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or (d) Limiting the use for publicity purposes of names of licensors or authors of the material; or (e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or (f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors. All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying. If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms. Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. **Termination.** You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11). However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation. Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice. Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. **Acceptance Not Required for Having Copies.** You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. **Automatic Licensing of Downstream Recipients.** Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License. An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts. You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. **Patents.** A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's “contributor version”. A contributor's “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License. Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version. In the following three paragraphs, a

"patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party. If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid. If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it. A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007. Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. **No Surrender of Others' Freedom.** If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.
13. **Use with the GNU Affero General Public License.** Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.
14. **Revised Versions of this License.** The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation. If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program. Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.
15. **Disclaimer of Warranty.** THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
16. **Limitation of Liability.** IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
17. **Interpretation of Sections 15 and 16.** If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

6. Acknowledgements

Prior to my starting Smoldyn, I attended an M-Cell workshop taught by Joel Stiles and Tom Bartoll, where I learned several of the concepts that became incorporated in the program. I started writing Smoldyn while I was a post-doc in Dennis Bray's laboratory at the University of Cambridge and funded by NIGMS grant GM64713. I made further additions during my next post-doc in Adam Arkin's laboratory at the Lawrence Berkeley National Laboratory, where I was funded by the Genomes to Life Project of the US Department of Energy and an NSF post-doctoral fellowship in biological informatics. Next, I made a lot of progress during my stay as a visiting scientist in Upi Bhalla's lab at the National Centre for Biological Research, in Bangalore, where I was funded by the Computer Research Laboratories (Pune, India). I made further additions as part of my work as a research fellow at the Molecular Sciences Institute, where I was funded by grants from the NIH and the MITRE corporation that were awarded to Roger Brent. Currently, I am a staff scientist in Roger Brent's laboratory at the Fred Hutchinson Cancer Research Center and funded by MITRE contracts to Roger Brent.

Many users and colleagues have proved invaluable comments and suggestions. These people include: Karen Lipkow, Dennis Bray, Adam Arkin, Upi Bhalla, Nathan Addy, Koichi Takahashi, Shahid Khan, Dylan Morris, Eric Cytrynbaum, David Savage, and Kevin Neff.