# Smoldyn Code Documentation

for Smoldyn version 2.56

Steve Andrews

# Contents

# Chapter 1

# Programmer's introduction

## 1.1 What is Smoldyn?

Smoldyn is a Brownian dynamics simulator. It represents space as a 1-, 2-, or 3-dimensional continuum, as opposed to a lattice, and it steps through time using finite length time steps. Smoldyn represents molecules as individual point-like particles and membranes as infinitely thin surfaces. Smoldyn simulates molecular diffusion, chemical reactions between individual molecules, and a wide variety of molecule-surface interactions. So far, Smoldyn has been used primarily for either detailed biophysics research problems, such as on diffusion-influenced reaction dynamics, or for investigating the effects of spatial organization on simple biological systems, such as the *Escherichia coli* chemotaxis system.

Smoldyn is also a community development project. I wrote nearly all of the core code, Jim Schaff and his colleagues at UCHC added some code for integrating Smoldyn into VCell, Nathan Addy wrote a rule-based modeling module called Libmoleculizer that almost worked, Lorenzo Dematte and Denis Gladkov independently parallelized Smoldyn to run on graphics processing units (GPUs), and Martin Robinson added adjacent-space hybrid simulation capabilities to Smoldyn.

In order to maintain and enhance Smoldyn's value for computational biologists, as opposed to letting it become a hodgepodge of mismatched code, it is helpful to carefully define what Smoldyn is, and what it isn't.

## 1.2 Smoldyn design philosophy

Central to Smoldyn's design philosophy is the concept of two distinct levels of approximation between physical reality and numerical simulation.

In the first level, physical reality is approximated to a perfectly defined *model system*. Smoldyn's model system builds on the one that Smoluchowski developed in 1917. It uses continuous space and continuous time. Depending on the context, molecules are represented as either point-like particles or as perfect spheres. In particular, they are (usually) point-like particles without excluded volume when considering diffusion and they are perfect spheres when considering chemical reactions. Model molecules do not have orientations, momenta, or kinetic energies because these aspects equilibrate in solution on much shorter time scales than the dynamics that the Smoluchowski model focuses on. As an example of the approximation from physical reality to the model system, molecular diffusion in reality is driven by a very high rate of intermolecular collisions and relies on van der Waals and steric molecular interactions. However, it is approximated in the model system as mathematically perfect Brownian motion, where each molecule behaves independently of every other molecule and moves with an infinitely detailed trajectory.

In the second level of approximation, the dynamics of the model system are approximated using numerical algorithms to yield the *simulated system*. In particular, Smoldyn approximates the model system through the use of finite simulation time steps. For example, Smoldyn simulates Brownian motion using Gaussian-distributed displacements at each time step. Because Smoldyn uses finite size time steps, it is nonsensical to ask about the state of the simulated system during time steps. Instead, the simulation produces what can

be seen as simulation system snapshots at the end of each time step. When the results in these snapshots are completely indistinguishable from those found with the ideal model system, to within computer round-off error, then the simulation algorithms are called "exact". Note that exactness only refers to agreement between the simulated system and the model system; correspondence between the model system and physical reality is a completely separate issue, and one which depends very much on the specific dynamics that the modeler wishes to investigate.

The Smoldyn software represents a balance between algorithms that are exact for the Smoluchowski model system and those that are computationally efficient. Often, these goals are actually complementary because highly accurate algorithms enable the use of long simulation time steps and hence enable fast simulations. However, there are also often tradeoffs, where better accuracy leads to slower simulations. The challenge with seeking a balanced approach between accuracy and computational efficiency is that the software users (i.e. modelers) generally aren't comfortable trusting software that is known to be inaccurate. For this reason, every algorithm in Smoldyn is exact in two ways. First, the simulated *rates* of all isolated algorithms are exact for any length time step. For example, in a simulation of the irreversible bimolecular reaction A + B $\rightarrow$ C, Smoldyn always gets the macroscopic reaction rate exactly correct, although the exact positions of the molecules are not necessarily in perfect agreement with those found for the model system. This focus on rates is important for Smoldyn to yield accurate equilibrium constants. Secondly, all simulated dynamics, regardless of how many algorithms are used in a simulation, approach exactness as simulation time steps are reduced towards zero. In the process, of course, simulation run times approach infinity, so exactness isn't actually achievable. However, having results that can approach exactness is important because it enables modelers to understand and quantify their simulation errors.

# Chapter 2

# Smoldyn code and build system

## 2.1 Subversion server

The Smoldyn code is hosted on a subversion server at the Fred Hutchinson Cancer Research Center (FHCRC), called `hedgehog.fhcrc.org`. Smoldyn is also hosted at SourceForge, but that site is only barely active, and thus is not recommended. Instead, most code development is done using the FHCRC server. Before you can check out code from the FHCRC server, you will need a login and password. Get those by e-mailing me (sandrews@fhcrc.org). Then, check out a copy of Smoldyn for working on with:

```
svn checkout https://hedgehog.fhcrc.org/users/sandrews/Smoldyn/trunknewdir
```

Here, `newdir` is the new local directory; it's optional, and I don't actually use it usually. You will need to enter a user name and a password. Your new directory will include the following files:

| file | author/origin | description |
|---|---|---|
| README | Steve | quick summary of what's in the directory |
| documentation | Steve | (directory) Smoldyn code documentation |
| examples | Steve | (directory) Smoldyn configuration files |
| Mac | Steve | (directory) Mac binaries and library files |
| source | Steve, Nathan | (directory) all source code |
| windows | Steve | (directory) Windows dlls, executables, and libraries |

## 2.2 Code merging

Some of the subversion tools merge code reasonably well. However, they aren't particularly informative. Here are some better options.

### Minimalist text based merging

`mydir="../../../gccCode/Library"`
: set variable

`diff Geometry.c $mydir`
: compares local version with version in $mydir, and prints out differences.

`grep -n 'Geo_Sphere_Normal' ../Smoldyn/source/*.c`
: finds all lines of Smoldyn source code that call Geo_Sphere_Normal.

### GUI applications

XCode offers FileMerge.app, which is at /Developer/Applications/Utilities/FileMerge.app. It's very easy to use.

Alternatively, Eclipse, at `www.eclipse.org` works well.

### svn merging

To keep two branches synchronized, they need to be merged periodically. For example, to merge the trunk changes into the nsv branch, one would normally change to the nsv root directory and then enter `svn merge https://hedgehog.fhcrc.org/users/sandrews/Smoldyn/trunk`. However, this won't acutally work because the FHCRC subversion server software is old. Thus, one has to specify version numbers. For example, enter `svn merge -r2043:2194 https://hedgehog.fhcrc.org/users/sandrews/Smoldyn/trunk`.

## 2.3 Source code dependencies

Code dependencies are shown below in a tree structure, such that the each file depends on the files that are indented below it. Note that the default Smoldyn configuration does not use either pthreads or Libmoleculizer, which results in many fewer code dependencies. Also, Smoldyn still builds and runs without its dependencies, but simply offers fewer features.

Smoldyn
- OpenGL
- libTiff
- zlib
- libiconv
- NSV
  - VTK

While some of the dependency code was included in the Smoldyn source distribution up to version 2.26, it is no longer included for versions 2.27 and newer versions. As a result, you need to get it yourself (or use a pre-compiled Smoldyn version). They are all fairly straightforward.

libtiff is from `http://download.osgeo.org/libtiff/`. I got version 3.9.6, which is the latest of the 3.9 series. Because I didn't want the zlib dependency of libtiff, I configured with "./configure –disable-zlib", and then entered "make" and "sudo make install" as usual.

libiconv is from `http://www.gnu.org/software/libiconv/`. I got version 1.14. I configured with "./configure –enable-static" so that I'd get the static library.

libXML++ is from `http://libxmlplusplus.sourceforge.net/`. While you can try to get a recent version, this is likely to be a major mistake because it has loads of dependencies, and those have dependencies, and so on. Instead, get libXML++ version 1.0.5. This is fully sufficient, and it works well. After downloading, extract the archive, change to the libxml++-1.0.5 directory, enter "./configure", "make", and "sudo make install". This was straightforward for me.

vtk is from `http://www.vtk.org/VTK/resources/software.html`. I downloaded version 5.10.1. To build it, I created subdirectory called "build" within the vtk download directory, changed to the build directory, and entered "cmake .." followed by "make". This is a very large package which took about a half hour to build. Lots of warnings were emitted, but the build completed successfully. Then, "sudo make install" installed the result. Quite a lot of files were installed.

For GPU Smoldyn, you will need some other things too. First is the CUDA library, which is from NVIDIA at `https://developer.nvidia.com/cuda-downloads`. This downloaded and installed itself. Next is the GLEW library (OpenGL Extension Wrangler library), which is from `http://glew.sourceforge.net`. This builds with simply "make" and "make install" (no "./configure" required). Next, the CUDPP library is from `http://code.google.com/p/cudpp/`. It builds with CMake (make a build directory, change to that directory, enter "cmake ..", "make", and "sudo make install"). For some reason, my build did not install

the cudpp_config.h file, so I had to do so by hand. From the CUDPP build directory, I entered "sudo cp ../include/cudpp_config.h /usr/local/include/" and that fixed problems.

The Boost library is from `http://www.boost.org/users/download/`. This doesn't get installed with an installer, but instead the whole directory gets copied over. It didn't work when I put it in a system location, but did work when I copied the "boost" subdirectory into the GPU code directory (Smoldyn/trunk/GPU/Gladkov/smoldyn-gpu-dg/).

## 2.4 Building with CMake (versions 2.27 and higher)

Smoldyn built using the GNU Autoconf, Automake, and Libtool tools through version 2.26. These GNU tools are remarkably arcane so I switched to CMake for versions 2.27 and above, which has been a vast improvement. I maintained the AutoTools documentation through version 2.31 and then removed it.

Building requires CMake, which can be downloaded from `http://www.cmake.org`. I use version 2.87, but any version above 2.8 should work. CMake installs trivially (at least on Mac), with a standard installer and no building required.

You can run CMake from either a command line interface (my preference) or with a GUI. At a command line interface, change directories to cmake. Every time you change CMake settings, you'll probably want to do a clean build. To do so, enter "rm -r *", while in the cmake directory (verify that you're in this directory!), to remove any prior build results. If you're asked about whether manifest.txt should be removed, say yes; this file shows the directories where Smoldyn was installed previously, thus providing information for you to remove it. For a default build, enter "cmake ..". A few test results will be printed out, and then configuring will be complete. See below for custom builds. The other option is to use the CMake GUI. It can be started by entering "cmake-gui" at a command line. Either way, when CMake is done, it will have written a lot of stuff to the cmake directory. Important files are "Makefile", which is the standard Makefile for the code and also smoldynconfigure.h, which is a C header file that the Smoldyn code uses for knowing what some important build parameters are.

Once configuring is complete, enter "make". Hopefully, Smoldyn will build, again with build files being put into the cmake directory. Finally, enter "sudo make install" and enter your password, to install Smoldyn to the usual place (/usr/local/bin on Linux and Mac systems).

For custom builds, you need to set various options to non-default settings. This is straightforward in the CMake GUI. There, you just check or uncheck boxes, as desired. Alternatively, from a command line interface, you can start CMake with "cmake .. -i" for interactive mode, and then CMake will ask you about each option. For each, you can just press return to select the default, or enter in values of your choice. Finally, you can also list each non-default option directly on the command line (preceded with a 'D', presumably for define).

Following are some helpful build options:

| Smoldyn option | default | effect when ON |
|---|---|---|
| `-DOPTION_VCELL` | `OFF` | Build for inclusion within VCell |
| `-DOPTION_NSV` | `ON` | Build with Next Subvolume support |
| `-DOPTION_PDE` | `OFF` | Build with support for PDE simulation |
| `-DOPTION_VTK` | `OFF` | Build with support for VTK visualization |
| `-DOPTION_STATIC` | `OFF` | Build using static libraries |
| `-DOPTION_USE_OPENGL` | `ON` | Build with graphics support |
| `-DOPTION_USE_LIBTIFF` | `ON` | Build with LibTiff support |
| `-DOPTION_TARGET_SMOLDYN` | `ON` | Build stand-alone Smoldyn program |
| `-DOPTION_TARGET_LIBSMOLDYN` | `OFF` | Build LibSmoldyn library |
| **CMake option** | **default** | **function** |
| `-DCMAKE_BUILD_TYPE` | `Release` | Choose CMake build type |
| options are: `None`, `Debug`, `Release`, `RelWithDebInfo`, and `MinSizeRel` | | |
| `-DCMAKE_CXX_COMPILER:FILEPATH` | clang | Compile with specific compiler |
| for example: /usr/bin/g++ | | |

## 2.5 Cross-compiling with MinGW

Smoldyn is cross-compiled for Windows from Mac using MinGW. Getting this set up can be a major challenge, but I'll describe what worked for me here. First of all, I got MinGW from MacPorts using "port install mingw-w64". This installed the meta-package mingw-w64, along with two sets of other packages: i686-w64-mingw32-... and x86_64-w64-mingw32-..., where the ... refers to the following 5 endings: binutils, crt, gcc, headers, and winpthreads.

I initially thought that the i686-w64-mingw32-gcc was the compiler that I was supposed to use. In fact, it sort of works, but not really. I wrote the standard hello.c, which compiled nicely on my Mac with "gcc -Wall hello.c -o hello". To cross-compile, I tried to compile with "i686-w64-mingw32-gcc -Wall hello.c -o hello.exe" but this returned the error message "i686-w64-mingw32-gcc: error trying to exec 'cc1': execvp: No such file or directory". However, this did work when I prefaced the instruction with "sudo" and then entered my password. This clearly implied that the issue had to do with permissions, but I was never able to figure out what was wrong and my query to Stack Overflow about this was useless.

When I finally switched to compiling with the x86_64-w64-mingw32-gcc compiler, using the line "x86_64-w64-mingw32-gcc -Wall hello.c -o hello.exe", then this worked perfectly without permission issues. I don't understand why the MacPorts MinGW download has the i686... and the x86... sets of files, which seem like reasonably parallel sets of files, of which only the x86... ones seem to be useful, but that's how it is.

Part of the MacPorts MinGW download is a lot of library code that's pre-compiled for MinGW. The useful portions seem to be in /opt/local/x86_64-w64-mingw32/include/ for the header files and /opt/local/x86_64-w64-mingw32/lib/ for the source files. These include most of the libraries that Smoldyn uses. However, it doesn't include GLUT. I downloaded the "freeglut 3.0.0 MinGW Package" from https://www.transmissionzero.co.uk/software/freeglut-devel/. The download is in the Smoldyn source directory, from which I copied the include and lib directory contents to the /opt/local/x86_64-w64-mingw32 directories.

I copied the i386 tiff*.h files from MinGW directory to /opt/local/x86...include directory. They seem ok, but they're clearly for a different architecture. I also tried the i386 libtiff.a static libtiff library, but Smoldyn building complained that it wasn't compatible. I also tried downloading libtiff.a from many different websites, but got the same result every time, that they weren't compatible. I'm giving up for now on offering tiff support for Windows.

## 2.6 Unit and regression testing

Smoldyn does not include unit tests in their purest form. Instead, I tested Smoldyn's algorithms, both for qualitative and quantitative performance, using simple Smoldyn configuration files, which function as unit tests. Many of these test files are in the examples directory. Individual files focus on specific algorithms, such as diffusion, unimolecular reactions, absorbing surface interactions, etc. Simply watching the simulation graphics is typically adequate for assessing qualitative performance, while data analysis and comparison against analytical theory is generally required for assessing quantitative performance.

The examples/S95_regression directory includes files used for regression testing. This directory includes many of the original unit tests, although typically modified for longer time steps, shorter total run times, and a fixed random number seed. In addition, I removed the original output for these unit tests and instead instructed them to output all molecule positions at the end of the simulation. The idea is that this is a very sensitive way of detecting whether all interactions during the simulation were the same between two runs, or not. This directory also includes a Python script regression.py. This script runs all unit tests, outputs their results to a subdirectory, and compares the results to those that there previously generated. All differences are reported, enabling detection of potential bugs.

To add a new unit test to the regression testing suite, simply make sure that the unit test runs relatively quickly (a few seconds) and has no text output. Then, copy and paste the top few lines from any of the current unit tests (e.g. bounce2.txt), so that the new unit test will output molecule positions. Finally, list the unit test name in the Python script.

# Chapter 3

# Files, macros, variables, etc.

The Smoldyn code is separated into several sets of files. (1) Library files, such as math2.c, are general-purpose C library files, nearly all of which I wrote. Smoldyn uses some of the functions in them, but far from all. (2) Each of these library files has a header, such as math2.h, that declares the structures and functions within that library file. These library files and headers are documented in separate documents, such as the file math2_doc.pdf. (3) The core Smoldyn source code is in files that begin with "smol", such as smolmolec.c. Smoldyn uses all of these functions. The main entry point to the program is in the file smoldyn.c, in the main function. This file also includes some high level functions for running the simulation. The other files take care of different portions of the simulation, such as molecules, virtual boxes, or surfaces. I have tried to encapsulate the code so that functions in any file are allowed to read directly from any structure, but only the functions in the file that corresponds to a structure is allowed to write to it. (4) The Smoldyn core source code files share a single header file, which is called smoldyn.h. It declares all data structures and function declarations. This header and the core Smoldyn files are documented here and in part I of the documentation.

## 3.1 Smoldyn source files

| file | function |
|---|---|
| smolboxes.c | virtual boxes |
| smolcmd.c | runtime interpreter commands |
| smolcomparts.c | compartments |
| smoldyn.c | top level functions |
| smoldyn.h | data structures and function declarations |
| smolgraphics.c | OpenGL graphics |
| smolmolec.c | molecules |
| smolport.c | ports |
| smolreact.c | reactions |
| smolsim.c | simulation structure and high level functions |
| smolsurface.c | surfaces |
| smolwall.c | walls |

## 3.2 Constants and global variables

**smoldyn.h**

`#define SMOLDYN_VERSION 2.16 // current Smoldyn version number`

This is the current version number of Smoldyn.

`#define DIMMAX 3`
: This is the maximum dimensionality permitted.

`#define VERYCLOSE 1.0e-12`
: Distance that is certain to be safe from round-off error during calculations.

`enum StructCond {SCinit,SCok,SCparams,SClists};`
: This is used in multiple structures to report the structure condition. SCinit is for just initialized, or initial initialization; SCok is for fully updated and ready for use; SCparams is for complete except that internal parameters need computation; and SClists is for structure lists and maybe also parameters need computation.

**smoldyn.c**

`simptr Sim;`
: Sim is a global variable for the current simulation structure. This is only used when graphics are being shown using OpenGL, because OpenGL does not allow variables to be passed in the normal way between functions.

## 3.3 Macros

`#define CHECK(A) if(!(A)) goto failure; else (void)0`
: This is a useful macro for several routines in which any of several problems may occur, but all problems result in freeing structures and leaving. Program flow goes to the label failure if A is false. Many people would consider both the use of a macro function and the use of a goto statement to be bad programming practice, and especially bad when used together. However, in this case it significantly improves code readability. As usual, partially defined structures should always be kept traversable and in good order so they can be freed at any time. The "else (void)0" termination of the macro is used so that if CHECK(...) is followed by an else, that else will refer to the prior if, and not to the CHECK. Because of the trailing else, compilers may complain if the CHECK macro follows an if and is not surrounded by braces.

`#define CHECKS(A,B) if(!(A)) strncpy(erstr,B,STRCHAR);goto failure; else (void)0`
: This is identical to CHECK, except that it also copies the included string to the variable erstr if a failure occurs. It is useful for error reporting.

## 3.4 Local variables

It has proven useful to use consistent names for local variables for code readability. In places, there are exceptions, but the following table lists the typical uses for most local variables. This table is also quite out of date.

| variable | type | use |
|---|---|---|
| `a` | `double` | binding radius for bimolecular reaction |
| `b,b2` | `int` | box address |
| `blist` | `boxptr*` | list of boxes, index is [b] |
| `boxs` | `boxssptr` | pointer to box superstructure |
| `bptr` | `boxptr` | pointer to box |
| `bval` | `double` | unbinding radius for bimolecular reaction |
| `c` | `int` | index of compartment |
| `ch` | `char` | generic character |
| `cmd` | `cmdptr` | pointer to a command |
| `cmds` | `cmdssptr` | pointer to the command superstructure |
| `cmpt` | `compartptr` | pointer to a compartment |

| | | |
|---|---|---|
| cmptss | compartssptr | pointer to a compartment superstructure |
| d | int | dimension number |
| dc1,dc2 | double | diffusion coefficients for molecules |
| dead | moleculeptr* | list of dead molecules, index [m] |
| difc | double* | list of diffusion coefficients, index [i] |
| dsum | double | sum of diffusion coefficients |
| dim | int | dimensionality of space |
| dt | double | time step |
| er | int | error code |
| erstr | char* | error string |
| face | enum PanelFace | face of a panel |
| flt1,flt2 | double | generic double variable |
| fptr | FILE* | file stream |
| got | int[] | flag for if parameter is known yet |
| i | int | molecule identity, reactant number, or generic integer |
| i1,i2,... | int | molecule identities |
| indx | int* | dim dimensional index of box position |
| itct | int | count of number of items read from a string |
| j | int | number of reaction for certain i |
| k | int | index of points within a compartment |
| lctr | int | line number counter for reading text file |
| line | char[] | complete line of text |
| line2 | char* | pointer to unparsed portion of string |
| live | moleculeptr** | list of live molecules, index [ll][m] |
| ll | int | index of live list |
| m,m2,m3 | int | index of molecule in list |
| m1,m2,m3 | int* | scratch space matrices of size dimxdim |
| mlist | moleculeptr* | list of molecules, index is [m] |
| mols | molssptr | pointer to molecule superstructure |
| mptr | moleculeptr | pointer to molecule |
| mptr1,mptr2 | moleculeptr | pointer to more molecules |
| ms | enum MolecState | molecule state |
| name | char** | names of molecules, index is [i] |
| nbox | int | number of boxes |
| ni2o | int | value of nidentôrder |
| nident | int | number of molecule identities |
| nl | int* | number of live molecules in a live list, index [ll] |
| nm,nm2 | char[] | name of molecule, reaction, or surface |
| nmol | int | number of molecules in list |
| npnl | int | number of panels for a surface |
| npts | int | number of points for a surface panel |
| nprod | int*,int | number of products for reaction [r] |
| nrxn | int* | number of reactions for [i] |
| nsrf | int | number of surfaces |
| o2 | int | order of second reaction |
| optr | int* | pointer to the order of reaction |
| order | int | order of reaction |
| p | int | reaction product number |
| p | int | panel number for surfaces |
| pfp | ParseFilePtr | configuration file pointer and information |
| pgemptr | double* | pointer to probability of geminate recombination |
| point | double** | list of points that define a panel [p][pt] |
| pos | double* | a position |
| prod | moleculeptr** | list of products for reaction [r], index is [p] |

| | | |
|---|---|---|
| pnl | panelptr | pointer to a panel |
| pnls | panelptr* | list of panels |
| ps | enum PanelShape | panel shape |
| pt | int | index for points, for surfaces |
| r | int | reaction number |
| r2 | int | reaction number for second reaction |
| rate | double* | requested rate of reaction [r] |
| rate2 | double* | internal rate parameter of reaction [r] |
| rate3 | double | actual rate of reaction |
| rev | int | code for reaction reversibility; see findreverserxn |
| rname | char** | names of reactions, index is [r] |
| rpar | double,double* | reversible parameter, index is [r] |
| rpart | char,char* | reversible parameter type, index is [r] |
| rptr | int* | pointer to reaction number |
| rxn | rxnptr | pointer to a reaction structure |
| rxn2 | rxnptr | pointer to a second reaction structure |
| s | int | surface number |
| side | int* | number of boxes on each side of space, index [d] |
| sim | simptr | pointer to simulation structure |
| smptr | simptr* | pointer to pointer to simulation structure |
| srf | surfaceptr | pointer to a surface structure |
| srfss | surfacessptr | pointer to a surface superstructure |
| step | double | rms step length of molecule or molecules |
| str1 | char[] | generic string |
| table | int** | table of reaction numbers for [i][j] |
| topd | int | top of empty molecules in dead list |
| total | int | total number of reactions in list |
| v1,v2,v3 | int* | scratch space vectors of size dim |
| w | int | index of wall |
| wlist | wallptr* | list of walls, index is [w] |
| word | char[] | first word of a line of text |
| wptr | wallptr | pointer to wall |

# Chapter 4

# Structures and functions

Smoldyn is written in C, with a C style. The proper maintenance of structures, which are described below, is a central aspect of the program. In general, the basic objects are molecules, walls, surfaces, and virtual boxes, each of which has its own structure. In many cases, these items are grouped together into superstructures, which are basically just a list of fundamental elements, along with some more information that pertains to the whole list. Reactions, compartments, and ports aren't really objects, but are also among the core data structures. Finally, a simulation structure is a high level structure which contains all the parameters and the current state of the simulation.

An aspect of structures that is important to note, especially if changes are made, is which structures own what elements. For example, a molecule owns its position vector, meaning that that piece of memory was allocated with the molecule and will be freed with the molecule. On the other hand, a molecule does not own a virtual box, but merely points to the one that it is in.

All allocation routines return either a pointer to the structure that was allocated, or `NULL` if memory wasn't available. Assuming that they succeed, all structure members are initialized, typically to 0 or `NULL` depending on the member type. All the memory freeing routines are robust in that they don't mind `NULL` inputs or `NULL` internal pointers. However, this is only useful and robust if allocation is done in an order that always keeps the structure traversable and keeps pointers set to `NULL` until they are ready to be initialized.

Both the code and the description below are sorted into categories: molecules, walls, reactions, boxes, surfaces, compartments, ports, and the simulation structure. In many cases, functions within each category work with only their respective object. However, the core program is highly integrated so that functions in one category may use objects in another category. Functions in one category (defined by the file that they are listed in) are not supposed to write to objects in other categories, although some exceptions may exist.

## 4.1 Header files

Smoldyn has several header files. They are: (*i*) smoldyn.h, which lists all of the structure declarations, (*ii*) smoldynfuncs.h, which lists all of the basic Smoldyn function declarations, (*iii*) libsmoldyn.h, which lists all declarations for Libsmoldyn, and (*iv*) smoldyn_config.h, which is automatically generated during the configuration process and which lists the compilation configure options.

## 4.2 Molecules (functions in smolmolec.c)

Each individual molecule is stored with a `moleculestruct` structure, pointed to by a `moleculeptr`. This contains information about the molecule's position, identity, and other characteristics that are specific to each individual molecule. These molecules are organized using a molecule superstructure, which contains lists of the active molecules, a list of unused molecule storage space called dead molecules, and other information about the molecules in general.

```
#define MSMAX 5
#define MSMAX1 6
```

```
enum MolecState {MSsoln,MSfront,MSback,MSup,MSdown,MSbsoln,MSall,MSnone};
enum MolListType { ,MLTport,MLTnone};
#define PDMAX 6
enum PatternData {PDalloc,PDnresults,PDnspecies,PDmatch,PDsubst,PDrule};
```

`MolecState` enumerates the physical states that a molecule can be in, which are respectively: in solution (i.e. not bound to a surface), bound to the front of a surface, bound to the back of a surface, transmembrane in the up direction, and transmembrane in the down configuration. `MSMAX` is the number of enumerated elements. While not a state that molecules are allowed to be in, `MSbsoln` is useful for reactions or surface actions to differentiate between in solution on the front of a surface versus on the back of a surface; MSsoln and `MSbsoln` are used for these, respectively. `MSMAX1` accounts for the additional destination state. Also, `MSall` and `MSnone` are useful enumerations for some functions to use as inputs or outputs.

`MolListType` enumerates the types of lists of molecules. `MLTsystem` is for molecules that are in the simulation system and `MLTport` is for molecule buffers for porting. `MLTnone` is not a type but is the absence of any type.

The `PatternData` enumeration, of which there are `PDMAX` options, are for the first `PDMAX` elements of the `mols->patindex` array, called the header. This array, as explained below, lists the species indices that match to text patterns. Each listing requires some information to describe how long the listing is, what's stored in it, and what it corresponds to, which is included in the header. These header elements should be retreived using the enumerated values. See below for their description and see the Wildcards section of the Code Design chapter.

```
typedef struct moleculestruct {
  long int serno;             // serial number
  int list;                   // destination list number (ll)
  double *pos;                // dim dimensional vector for position [d]
  double *posx;               // dim dimensional vector for old position[d]
  double *via;                // location of last surface interaction [d]
  double *posoffset;          // position offset arising from jumps [d]
  int ident;                  // species of molecule; 0 is empty (i)
  enum MolecState mstate;     // physical state of molecule (ms)
  struct boxstruct *box;      // pointer to box which molecule is in
  struct panelstruct *pnl;    // panel that molecule is bound to if any
  struct panelstruct *pnlx;   // old panel that molecule was bound to if any
  } *moleculeptr;
```

`moleculestruct` is a structure used for each molecule. `serno` is the unique serial number that each live molecule is given; it is assigned by the utility function `getnextmol`, which should be used to add new live molecules to the system. If a molecule is being imported from elsewhere, it is legitimate to overwrite the serial number with the previous value. `list` is the master list number that the molecule should be listed in (-1 for dead, other values for live lists); this is modified with `molkill`, `molchange`, or one of the `addmol` functions. `pos` and `posx`, both of which are owned by the structure, are always valid positions, although not necessarily within the system volume. `posx` is the position from the previous time step, used to determine if a molecule crossed a wall or surface. `posoffset` is the cumulative position offset that should be added to the position to correct for jumps and periodic boundaries (it replaced the `wrap` element). `via` is the location of the most recent surface interaction. `ident` should always be between 0 and `nident-1`, inclusive. A molecule type of 0 is an empty molecule for transfer to the dead list (and should also have `list` equal to -1). Except during set up, `box` should always point to a valid box. `mstate` is the physical state of the molecule, which might be solvated or any of several surface-bound positions.

If this molecule is bound to a surface, `pnl` points to that surface panel. Also, `pnlx` points to the panel that `posx` was on.

```
typedef struct molsuperstruct {
  enum StructCond condition;    // structure condition
  struct simstruct *sim;        // simulation structure
  int maxspecies;             // maximum number of species
  int nspecies;               // number of species, including empty mols.
  char **spname;              // names of molecular species
```

```
int maxpattern;              // maximum number of patterns
int npattern;                // actual number of patterns
char **patlist;              // list of patterns [pat]
int **patindex;              // species indices for patterns [pat][j]
char **patrname;             // pattern reaction name if any [pat]
double **difc;               // diffusion constants [i][ms]
double **difstep;            // rms diffusion step [i][ms]
double ***difm;              // diffusion matrix [i][ms][d]
double ***drift;             // drift vector [i][ms][d]
double *****surfdrift;       // surface drift [i][ms][s][ps][d]
double **display;            // display size of molecule [i][ms]
double ***color;             // RGB color vector [i][ms]
int **exist;                 // flag for if molecule could exist [i][ms]
moleculeptr *dead;           // list of dead molecules [m]
int maxdlimit;               // maximum allowed size of dead list
int maxd;                   // size of dead molecule list
int nd;                     // total number of molecules in dead list
int topd;                   // index for dead list; above are resurrected
int maxlist;                 // allocated number of live lists
int nlist;                   // number of live lists
int **listlookup;            // lookup table for live lists [i][ms]
char **listname;             // names of molecule lists [ll]
enum MolListType *listtype; // types of molecule lists [ll]
moleculeptr **live;          // live molecule lists [ll][m]
int *maxl;                   // size of molecule lists [ll]
int *nl;                     // number of molecules in live lists [ll]
int *topl;                   // live list index; above are reborn [ll]
int *sortl;                  // live list index; above need sorting [ll]
int *diffuselist;            // 1 if any listed molecs diffuse [ll]
long int serno;              // serial number for next resurrected molec.
int ngausstbl;               // number of elements in gausstbl
double *gausstbl;            // random numbers for diffusion
int *expand;                 // whether species expand with libmzr [i]
long int touch;              // counter for molecule modification
} *molssptr;
```

`molsuperstruct` contains and owns information about molecular properties and it also contains and owns lists of molecules. `condition` is the current condition of the superstructure and `sim` is a pointer to the simulation structure that owns this superstructure. `maxspecies` is the number of molecular species for which the arrays are allocated, `nspecies` is the actual number of defined species, and `spname` is the list of names for those species. Other superstructures that have `maxspecies` elements, such as surfaces, are intended to be copies of the one here for internal use, while this one remains the master.

The definitive version of the pattern stuff is described in the Wildcards section of the Code Design chapter. However, a little is described here, too. Patterns represent one or more species names. Each string that is used gets recorded here. This list is only updated when it is used. `maxpattern` slots are allocated for patterns, of which `npattern` are actually used. The actual list of patterns is called `patlist`; these are sorted by alphabetical order. Sorted in parallel are the lists `patindex` which is a list of lists, and `patrname`, which is a list of reaction names. `patrname` is only used if the pattern represents a reaction, and is used to differentiate between multiple different reactions that have identical patterns. In `patindex`, the outer list corresponds to the patterns. In the sublists, the first several elements, called the header, give important information about the list. The header occupies the first `PDMAX` list elements.

Diffusion is described with `difc`, a list of diffusion constants; `difstep` is a vector of the rms displacements on each coordinate during one time step if diffusion is isotropic; and `difm` is a list where each element is either a `NULL` value if diffusion is isotropic or a `dimxdim` size diffusion matrix (actually the square root of the matrix). `drift` is the vector for molecular drift, relative to system coordinates. `surfdrift` is for molecular drift relative to the local surface panel coordinates; this memory is only allocated as required. All of these are arrays on the molecule identity, followed by arrays on the molecule state (size `MSMAX`). `display` is simply

the size of molecules for graphical output (which scales differently for different output styles) and `color` is the 3-dimensional color vector for each molecule; again size of sate list is `MSMAX`.

`exist` is 1 for each identity and state that could be a part of the system and 0 for those that are not part of the system. This is set in `molsupdate`, were any molecules and states that exist then are recorded, as are all reaction products. If commands create molecules, they should also set the exist flag with `molsetexist`.

`expand` is a flag for on-the-fly rule-based modeling. It is initialized to 0 and stays that way so long as there have never been any molecules of this species, it is increased to 1 if at least one molecule of this species has been created but it has not yet been used for rule expansion, and is set to either 2 or 3 if it has been used for expansion.

`touch` is a counter that counts the number of times that the list of molecules has been modified. No meaning is ascribed to any particular value. Instead, it can be used to determine if the molecule state has changed between one call of a function and another call of a function, used to prevent recomputing things if it hasn't changed. The `touch` value should be incremented by any function that directly changes molecules, whether it creates new ones, kills existing ones, or moves them. Functions that call other functions for these purposes (e.g. that call `addmol`, `molkill`, or `molchangeident`) do not increment `touch`. Molecules are not considered to be changed if they are merely re-sorted between molecule lists or re-assigned to boxes.

The molecule lists are separated into two parts. The first set is the live list, which are those molecules that are actually in the system or that are being stored for transfer elsewhere (i.e. buffers for ports are also live lists); the others are in the dead list, are empty molecules, and have no influence on the system. If more molecules are needed in the system than the total number allocated, the program sends an error message and ends; in the future, it may be possible to dynamically create larger lists. Upon initialization, all molecules are created as empty molecules in the dead list, whereas during program execution, all lists are typically partially full. After sorting, each live list, `ll`, has active molecules from element 0 to element `nl[ll]-1`, inclusive, and has undefined contents from `nl[ll]` to `maxl[ll]-1`. Similarly, the dead list is filled with empty molecules from 0 to `nd-1`, and has undefined contents from `nd` to `maxd-1`; in this case, `topd` equals `nd`.

Functions other than `molsort`, such as chemical reactions, are allowed to kill live molecules (with `molkill`) or resurrect dead ones (with `getnextmol`, `addmol`, or `addsurfmol`) but they should not move molecules or change the list indices. With new molecules that are gotten with `getnextmol`, set the molecule identity, state, list (with `mols->listlookup`), position, old position, panel if appropriate, and box. Set the `box` element of the molecule to point to the proper box, but do not add the molecule to that box's molecule list. It is now in the resurrected list, which is the top of the dead list between `topd` and `nd-1`, inclusive. Routines should be written so that these mis-sorted molecules do not cause problems. They are sorted with `molsort`, which moves the empty molecules in the live lists to the dead list, moves the resurrected ones to the top of the proper live list, compacts the live lists (molecule order is not maintained), and identifies the newly reborn molecules in the live lists by setting `topl[ll]`; the reborn molecules extend from `topl[ll]` to `nl[ll]`.

Example of the lists:

| index | live[0] | live[1] | dead |
|---|---|---|---|
| 8 | ? | ? | maxd ? |
| 7 | maxl[0] ? | ? | - |
| 6 | - | ? | - |
| 5 | - | maxl[1] ? | - |
| 4 | - | - | - |
| 3 | nl[0] - | - | nd - |
| 2 | 2 | topl[1],nl[1] - | topd 1 |
| 1 | topl[0] 1 | 0 | 0 |
| 0 | 0 | 3 | 0 |

Here, each list has `max`=8, and so is indexed with `m` from 0 to 7. A '?' is memory that is not part of that which was allocated, a '-' is a `NULL` value, a '0' is an empty molecule, and other numbers are other identities ('1' and '2' are mobile, whereas '3' is immobile). The '0's in the the two live lists are to be transferred to the dead list during the next sort, while the '1' in the dead list has been resurrected and is to be moved to mobile live list. Based on the `topl[0]` index, it can be seen that the '1' and '2' in the mobile live list were

just put there during the last sorting, and so are reborn molecules.

There is one dead list and there are `nlist` live lists. The dead list has total size `maxd` and is filled to level `nd`. The index `topd`, which is between 0 and `nd`, separates the dead molecules (list element is -1) with indices from 0 to `topd-1` from the resurrected molecules (list elements 0) that have indices between `topd` and `nd-1`. The dead list is automatically expanded up to size `maxdlimit` if this value is positive, and is automatically expanded without limit if `maxdlimit` is negative (the default). Live list `ll` has total size `maxl[ll]` and is filled to level `nl[ll]`. The index `topl[ll]`, which is between 0 and `nl[ll]` separates the molecules that were there on the prior time step that have smaller indices from those that were created in the last time step, called the reborn molecules, which have higher indices.

**enumerated type functions**

`enum MolecState molstring2ms(char *string);`
Returns the enumerated molecule state value, given a string input. Permitted input strings are "solution", "aq" (aqueous), "front", "back", "up", "down", "fsoln", "bsoln", and "all". Returns MSnone if input is "none" or is not recognized.

`char *molms2string(enum MolecState ms,char *string);`
Returns the string that corresponds to the enumerated molecule state `ms` in string, which must be pre-allocated. Also, the address of string is returned to allow for function nesting.

`enum MolListType molstring2mlt(char *string);`
Returns the enumerated molecule list type, given a string input. Permitted input strings are "system" and "port". Returns `MLTnone` for all other input.

`char *molmlt2string(enum MolListType mlt,char *string);`
Returns the string that corresponds to the enumerated molecule list type `mlt`. The string needs to be pre-allocaed; it is returned to allow function nesting.

**low level utilities**

`int molismobile(simptr sim,int species,enum MolecState ms);`
Returns 1 if molecules of species `species` and state `ms` are mobile at all and 0 if they are not. Mobility includes isotropic and anisotropic diffusion, drift, and surface drift. `MSbsoln` is an allowed input, which always returns the same result as a `MSsoln` input.

`int molstring2pattern(const char *str,enum MolecState *msptr,char *pat,int mode);`
This assembles a species pattern string from molecule species strings. For example, if the user enters a reaction as `A*(front) + B|C(soln) -> A*{B|C}(front)`, then the function that reads this string (which are `molstring2index1` and `rxnparsereaction`) will send the `A*(front)`, `B|C(soln)`, and `A*{B|C}(front)` strings to this function for them to be assembled in the pattern `A*+B|C\nA*{B|C}`, and for the states to be returned to the calling function. The pattern is then generally sent off to `molpatternindex`.

Enter `str` as a string of which the first word is the species name or pattern to be processed. The state, if listed at the end of the name string, is returned in `msptr`. Enter `mode` as 0 if this is a new pattern, 1 if the current text being added to the pattern is part of the "match" side of the pattern (the first two words in the above example), or 2 if the current text being added to the pattern is part of the "substitute" side of the pattern (the last word in the above example). The `pat` string needs to allocated beforehand to size `STRCHAR`.

Returns 0 for success, -1 if `str` or `pat` are missing or if there is no first word in `str`, -2 if the parentheses in `str` don't match up, -3 if the molecule state could not be read, or -4 if the pattern length exceeds the maximum number of allowed characters (which is `STRCHAR`).

`int molreversepattern(simptr sim,const char *pattern,char *patternrev);`
Takes in a reaction pattern in `pattern` and reverses it for the reverse reaction, returning the result in

the string `patternrev`. This only works for patterns that represent reactions. This simply writes the reverse pattern as the products of the original and then the reactants. Returns 0 for success or -1 if `pattern` is not a reaction.

`int molpatternindex(simptr sim,const char *pattern,const char *rname,int isrule,int update,int **indexptr);`

This function takes in a pattern string, in `pattern`, and returns the list of species or species combinations that correspond to this pattern. To make this function efficient, it records its answers in the molecule superstructure `patlist` and `patindex` lists, so that the list does not need to be recomputed if it is asked for multiple times. The returned list of species is the proper `patindex` list from the molecule superstructure, including its header data. The data returned by this function is a pointer to the original data, not a copy of it, so it should generally not be modified. See the pattern discussion under the molecule superstructure description for details about the data that are returned. Enter `isrule` with 1 if this is a rule, in which case new species names that arise from expanding the pattern get added to the simulation (but not their reactions), or with 0 if this is not a rule, meaning that new species names are errors. Enter `update` with 0 if the index list should not be updated at all, with 1 if the index list should be created if it doesn't exist already, and with 2 if the index list should be updated to the current list of species (or to the recently created species for reactions and on-the-fly generation).

This function does not support species groups that are defined without wildcards.

Returns 0 for success, -1 for inability to allocate memory, -2 if no wildcards were entered and one or more of the match species is unknown, -3 if a substitute species is unknown and the substitute species had no wildcards in it, -4 if the match string included more words than allowed by this function (which is 4 currently), -5 if a trial match string was too long to fit in `STRCHAR` characters (even if this wasn't actually a match), -6 if species generation failed, -11 for inability to allocate memory, -12 for missing ' ' operand, -13 for missing & operand, -15 for mismatched braces, or -20 for a destination pattern that is incompatible with the matching pattern (i.e. it has to have either 1 destination or the same number of destination options as pattern options). If `update` is set to 0, then no errors are possible. In this case, if the pattern is not in the list, then the function does not add it to the list, but simply returns a value of 0 and `indexptr` pointing to `NULL`. If `update` is set to 1, then the only error possible is -1, for inability to allocate memory.

First, this function looks to see if `pattern` is already in the pattern list. If the pattern is not found, this function adds it, while maintaining alphabetical order. This may require expanding the list of patterns and pattern indices if the list is full.

Next, for both new and old patterns, this function determines if the indices are up to date, meaning that the `PDnspecies` value is negative, which means that it never needs updating, or it is equal to the current number of species in the simulation, which means that it is already up to date. If the indices are up to date, then the function is done; it simply returns the correct index list. If not, then the function figures out the header to the `index` list, if necessary, and prepares several variables for later use. These variables are: `matchstr`, `substr`, and `newline`, which refer to the pattern string; `istart` and `jstart`, which are the species number to start updating from and the starting result in the index list to start updating to; `matchwords`, `subwords`, and `totalwords`, which are the numbers of each type of word in the pattern; `haswildcard` and `hasspeciesgroup`, which are whether the pattern has wildcard characters and/or a species group; and `nspecies` which is the current number of species in the simulation.

Finally, the function goes through a number of pattern types, from simplest to most general. The simple ones are simply special cases of the most general one, but they are included because their code will run much faster when applicable, and because the simple cases help to make sense of the general case.

(1) If the pattern is "all", then the list should include all species names, but there's nothing else to worry about.

(2) If the pattern has no newline, no wildcards, and 1 matchword, then it must be just the name of a species. It could also be the name of a species group, but it shouldn't be because those are set for

updating not required. Assuming it's a single species, this makes space for it in the index, finds the identity value of the species, sticks it in the index, and sets the header values. If the function didn't find the species name (or if the pattern is a species group name), then this returns error code -2 to indicate an unknown species name.

(3) If the pattern has no newline character and one matchword, then it must be a single species name with a wildcard character. If that's the case, then the function goes through all species that haven't been considered before, sees if each one matches to the match word, and adds them to the `index` list if so.

(4) Any other patterns without newline characters, meaning those that have multiple matchwords, are not allowed, so they result in an error.

(5) Next, the function considers patterns with a newline but no wildcards and no species groups, and one matchword. In this case, it is for a reaction with exactly one reactant. It will also have exactly one entry in the `index` list. In this case, the function reads through the `matchwords` to get each reactant name and puts those in `ispecies`. Then, typically, it keeps on going, putting the reactant identities in the `index` variable. Then, it reads through `subwords` to get the product names, which it converts to identities, and puts those in `index`. The `PDnspecies` value is set to -1 because this `index` should never need updating again. This function can also handle rules, possibly on-the-fly. For example, a reaction rule could be A + B $\rightarrow$ C. If this is generated on-the-fly, then it simply says that species C should not be created until there is a molecule of A or B.

(6) Next, the same thing but 2 matchwords.

(7) Next, with newline, species group or wildcards, and 1 matchword.

(8) Finally, with newline, species group or wildcards, and 2 matchwords.

```
int molstring2index1(simptr sim,char *str,enum MolecState *msptr,int **indexptr);
```

This reads the first word of the string in `str`, parses it to find the state listed, if any, and determines what species name or names it refers to. The state is returned in `msptr` and the species index list is pointed to by `indexptr`. The function simply calls `molstring2pattern` and then `molpatternindex` sequentially. On success, if there is exactly one result in the list and exactly one match item in the pattern, this returns the index of the result. If there are multiple results, it returns 0, which is still a successful result.

It can also return the following error codes: -1 if `str` is missing or has no first word, -2 if the parentheses in `str` don't match up, -3 if the molecule state could not be read, -4 if there are no wildcards in the string and the species name is unknown (or if the number of characters in `str` is more than the maximum allowed in a pattern (256), which shouldn't ever happen), -5 if the species is "all", -6 if the logic expansion failed due to missing & operand or mismatched braces, or -7 if memory could not be allocated.

```
int moladdspeciesgroup(simptr sim,char *group,char *species,int imol);
```

Creates a species group named `group` and adds the species (or species group, including species names with wildcards) named `species` to that group. If the group already exists, then this just adds the species to the existing group. The `species` value is allowed to be `NULL` for creating an empty group and it is also allowed to be a group as well, including a group specified using wildcard characters. This function uses the same pattern infrastructure as for wildcard characters, storing the data in the same `patstring` and `patindex` data structures. Also, `species` can be `NULL` and a single species can be added instead using `imol`, where this is the identity of a single species.

Returns 0 for success, -1 if the group name name is missing, -2 if there are parentheses mismatches (which shouldn't be there anyhow), -3 if a molecule state could not be read (which shouldn't be there), -4 if the species name does not correspond to any species, -5 if the group name is "all", -6 if logic expansion failed due to missing & operand or mismatched braces, -7 if memory could not be allocated, -8 if a molecule state is given and isn't "all", or -9 if the group name is the same as an existing molecule name.

`char *molpos2string(simptr sim,moleculeptr mptr,char *string);`
Writes molecule position in `mptr->pos` to `string` using "%g" formatting code for `sprintf`. Each coordinate value, including the first one, is preceded by a space. If the simulation includes surfaces, this function ensures that the written position, including round-off errors, is both in the same box and on the same side of all surface panels (not including the panel that the molecule is bound to, if any) as the actual position. If this function cannot achieve these criteria after 50 attempts, it prints a warning, and returns the string.

`void molchangeident(simptr sim,moleculeptr mptr,int ll,int m,int i,enum MolecState ms,panelptr pnl);`
Changes the identity or state of a molecule that is currently in the system to species `i` and state `ms`. It is permissible for `i` to equal 0 for the molecule to be killed, which is equivalent to calling `molkill`. `mptr` is a pointer to the molecule and `ll` is the list that it is currently listed in (probably equal to `mptr->list`, but not necessarily). If the molecule's list information should be updated to reflect the identity change, which is nearly always, then `ll` should be the molecule's list; if not though, then send in both `ll` and `m` as -1. If `m` is known, then enter the index of the molecule in the master list (i.e. not a box list) in `m`; if it's unknown set `m` to -1. If the molecule is to be bound to a panel (independent of whether it was bound to a panel before or not), enter the panel in `pnl`. Or, if it is to be in solution but adjacent to a panel, enter this panel in `pnl`.

This function sets some parameters of the molecule structure, fixes the location as needed, and, if appropriate, updates `sortl` to indicate to `molsort` that sorting is needed. This also increments the molecule `touch` value to show that molecules have been touched.

`int molssetgausstable(simptr sim,int size);`
Sets the size of the Gaussian look-up table to `size` and also allocates the table, if needed. Setting `size` to 0 or a negative number keeps the current size if it has already been allocated, or creates a table with the default size (4096) if not. Otherwise, `size` is required to be an integer power of two. This will replace an existing table if the new size if different from the previous one. Returns 0 for success, 1 for insufficient memory, or 3 if the size is not an integer power of two.

`void molsetdifc(simptr sim,int ident,int *index,enum MolecState ms,double difc);`
Sets the diffusion coefficient for molecule `ident` and state `ms` to `difc`. For multiple identities, enter them in `index` using the pattern index header. If `ms` is `MSall`, this sets the diffusion coefficient for all states. This does not update rms step sizes or reaction rates.

`int molsetdifm(simptr sim,int ident,int *index,enum MolecState ms,double *difm);`
Sets the diffusion matrix for molecule `ident` and state `ms` to `difm`. Any required matrices that were not allocated previously are allocated here. For multiple species, enter them in `index` using the pattern index header. If `ms` is `MSall`, this sets the diffusion matrix for all states. This returns 0 for successful operation and 1 for failure to allcate memory. This updates the isotropic diffusion coefficient but does not update rms step sizes or reaction rates.

`int molsetdrift(simptr sim,int ident,int *index,enum MolecState ms,double *drift);`
Sets the drift vector for molecule `ident` and state `ms` to drift. Any required vectors that were not allocated previously are allocated here. For multiple species, enter them in `index` using the pattern index header. If `ms` is `MSall`, this sets the drift vector for all states. This returns 0 for successful operation and 1 for failure to allocate memory.

`int molsetsurfdrift(simptr sim,int ident,int *index,enum MolecState ms,int surface,enum PanelShape ps,double *drift);`
Sets the surface drift vector for molecule `ident`, state `ms`, surface `surface`, and panel shape `ps` to `drift`. Any required memory that was not allocated previously is allocated here. For multiple species, enter them in `index` using the pattern header. If `ms` is `MSall`, this sets the surface drift vector for all surface-bound states. In addition, `surface` can be -1 to indicate all surfaces and `ps` can be `PSall` to indicate all panel shapes. Any combination of "all" conditions is permitted. This returns 0 for successful operation and 1 for failure to allocate memory.

`void molsetdisplaysize(simptr sim,int ident,int *index,enum MolecState ms,double dsize);`
Sets the display size for molecule `ident` and state `ms` to `dsize`. For multiple species, enter them in `index` using the pattern header. If `ms` is `MSall`, this sets the display size for all states.

`void molsetcolor(simptr sim,int ident,int *index,enum MolecState ms,double *color);`
Sets the color for molecule `ident` and state `ms` to the 3-dimensional RGB vector `color`. For multiple species, enter them in `index` using the pattern header. If `ms` is `MSall`, this sets the color for all states.

`void molsetlistlookup(simptr sim,int ident,int *index,enum MolecState ms,int ll);`
Sets the list lookup table value to live list number `ll` for molecule `ident` and state `ms`. For multiple species, enter them in `index` using the pattern header. Special codes are also possible in the `ident` input: `ident=-7` implies all diffusing molecules, and `ident=-8` implies all non-diffusing molecules. Using `ms=MSall` implies all states. Note that the `listlookup` element is defined for both `MSsoln` and `MSbsoln`, and they are always set to the same values.

`void molsetexist(simptr sim,int ident,enum MolecState ms,int exist);`
Sets the `exist` element of the molecule superstructure for identity `ident` and state `ms` to `exist`; "all" inputs are not permitted.

`int molcount(simptr sim,int i,enum MolecState ms,int max);`
Counts the number of molecules of type `i` and state `ms` currently in the simulation. If `max` is -1 it is ignored, and otherwise the counting stops as soon as `max` is reached. Either or both of `i` and `ms` can be set to "all"; enter `i` as a negative number and enter `ms` as `MSall`. All molecule lists and the dead list are checked; porting lists are included. This function returns correct molecule counts whether molecule lists have been sorted since recent changes or not. It runs fastest if molecule lists have been sorted. If `i` is less than zero, this implies all species; if `i` is greater than zero, this implies that specific species; and if `i` equals zero, this implies that the `index` entry should be used instead. In this last case, enter lists of species using `index`, using the standard index pattern header.

This function is essentially the same exact thing several times in a row, for the different input cases and with slightly different outer loops. This could be substantially shortened, but is done this way for better speed. This function requires that the `index` list be sorted.

`void molscancmd(simptr sim,int i,int *index,enum MolecState ms,cmdptr cmd,enum CMDcode(*fn)(simptr,cmdptr,char*));`
Scans over all molecules that meet the criteria listed and calls the function `fn` for each one. This function is very similar to `molcount`, except that it calls a function for each molecule rather than just counting them. As with `molcount`, this checks all molecule lists and the dead list; it also works whether molecules have been sorted or not. Send in `i` as the species number, which should be less than 0 for all species, 0 to indicate that `i` should be ignored and `index` should be used instead, or as a positive number for that specific species. Enter `index` as a sorted list of species, using the standard index pattern header and format; it is only used if `i` equals zero. Enter `ms` as `MSall` for all species and as a specific species otherwise. The function entry is designed for a Smoldyn command function and so has the same format, despite the fact that this format isn't really ideal in this case. The function gets passed the simulation pointer in the first argument, `cmd` in the second argument, and a pointer to the current molecule in the third parameter (cast to a `char*` due to the non-ideal formatting). The function should return `CMDok` to indicate that the scan should continue or `CMDstop` to indicate that the scan should stop. See the commands section of the manual to see how this function can be used.

`void molscanfn(simptr sim,int i,int *index,enum MolecState ms,char *erstr,double(*fn)(void*,char*,char*);`
This is identical to `molscancmd` but has a slightly more versatile function declaration that's not designed just for commands. The `erstr` string is for returning errors. The call-back function, `fn` should take the simulation structure as its first argument, the error string as its second argument, and the molecule pointer, cast as a `char*`, as its third argument.

`int molismatch(moleculeptr mptr,int i,int *index,enum MolecState ms);`
Tests to see if the molecule in `mptr` matches the conditions given in species number `i`, index list `index`,

and state `ms`. These latter three elements should be the values returned by `molstring2index1`, so they can be for single species, species groups, etc.

`int MolCalcDifcSum(simptr sim,int i1,enum MolecState ms1,int i2,enum MolecState ms2);`
Calculate and returns diffusion coefficient sums. This allows `ms1` and/or `ms2` to be the `MSbsoln` state. Also, enter `i1` and/or `i2` as 0 to not include it in the sum.

**memory management**

`moleculeptr molalloc(int dim);`
`molalloc` allocates and initiallizes a new `moleculestruct`. The serial number is set to 0, the `list` to -1 (dead list), positional vectors to the origin, the identity to the empty molecule (0), the state to `MSsoln`, and `box` and `pnl` to `NULL`. The molecule is returned unless memory could not be allocated, in which case `NULL` is returned.

`void molfree(moleculeptr mptr);`
`molfree` frees the space allocated for a `moleculestruct`, as well as its position vectors. The contents of `box` and `pnl` are not freed because they are references, not owned by the molecule structure.

`molexpandsurfdrift(simptr sim,int oldmaxspec,int oldmaxsrf);`
Expands the surface drift data structure, when the species list and/or the surface list is expanded. Enter `oldmaxspec` and `oldmaxsrf` with the maximum number of species and surfaces before expansion (if only one needs to be expanded, then both still need to be listed, but one will match the current maximum). This function simply calls `molsetsurfdrift` with all of the data in the current data structure, which re-builds the data structure in a larger format. This function is called by `surfacessalloc` and `molssalloc`.

`void molfreesurfdrift(double *****surfdrift,int maxspec,int maxsrf);`
Frees the space allocated for all surface drift data, which is stored in a molecule superstructure.

`int molpatternindexalloc(int **indexptr,int n);`
Allocates space for a single index list of the species pattern string lookup table, or expands it. If space is needed for a new index list, send in `indexptr` pointing to `NULL`; if an existing index list needs expansion, send in `indexptr` pointing to the start of the list that needs expansion. Send in `n` as the number of total spaces (including the overhead spaces) that are desired, or send in `n` as -1 for automatic allocation and expansion. This allocates memory and sends the result back pointed to by `indexptr`. Returns 0 for success or 1 for inability to allocate memory. This sets element 0 of the result to equal the allocated length of the array. It sets all other newly created elements to 0.

`int molpatternalloc(simptr sim,int maxpattern);`
Allocates space for species patterns and their indices. Send in `sim` as the simulation structure and `maxpattern` for the desired total allocated number of pattern spaces. This allocates space, copies over any existing data, and initializes the new spaces to empty values. This takes care of the `patlist` and `patindex` lists.

`molssptr molssalloc(molssptr mols,int maxspecies);`
`molssalloc` allocates and initializes a molecule superstructure. This function may be called multiple times, in order to increase the maximum number of species. The Gaussian table is left empty; it is filled in in `molsupdate`. Returns `NULL` if there is insufficient memory. Enter `maxspecies` with your desired number of simulated species. One more than this will actually be allocated because this assigns species number 0 to the "empty" species.

`int mollistalloc(molssptr mols,int maxlist,enum MolListType mlt);`
Allocates `maxlist` new live lists of list type `mlt` for the already existing molecule superstructure `mols`. This works whether there were already were live lists or not. Returns the index of the first live list that was just added for success or a negative code for failure: -1 for out of memory, -2 for a negative `maxlist` input value, or -3 for a `NULL` `mols` input. The `maxlist` element of the superstructure is updated. The `nlist` element of the superstructure is unchanged.

This does all of the allocation separately from the molecule superstructure. At the end, if all goes well, it frees the current memory and replaces it with the new memory.

`int molexpandlist(molssptr mols,int dim,int ll,int nspaces,int nmolecs);`
Expands molecule list, where `mols` is the molecule superstructure and `dim` is the system dimensionality. This both creates new lists or expands existing lists, as required. If `ll` is negative, the dead list is expanded and otherwise live list number `ll` is expanded. If `nspaces` is negative, the list size is doubled and otherwise `nspaces` spaces are added to the list. The first `nmolecs` of these spaces are filled with new dead molecules (`mptr->list` element set to -1). Because this shouldn't normally be called with `ll`$\geq$0 and `nmolecs`>0, error code 2 is returned if this happens. This returns 0 for success, 1 for out of memory during list expansion, 2 for illegal inputs, 3 for more molecules are being created than will fit in the list even after expansion, and 4 for out of memory during molecule allocation.

`void molssfree(molssptr mols,int maxident,int maxsrf);`
`molssfree` frees both a superstructure of molecules and all the molecules in all its lists.

**data structure output**

`void molssoutput(simptr sim);`
`molssoutput` prints all the parameters in a molecule superstructure including: molecule diffusion constants, rms step lengths, colors, and display sizes; and dead list and live list sizes and indices.

`void writemols(simptr sim,FILE *fptr);`
Writes all information about the molecule superstructure to the file `fptr` using a format that can be read by Smoldyn. Does not write information about individual molecules. This allows a simulation state to be saved.

`void writemolecules(simptr sim,FILE *fptr);`
Writes information about all individual molecules to the file `fptr` using a format that can be read by Smoldyn. This allows a simulation state to be saved.

`int checkmolparams(simptr sim,int *warnptr);`
Checks some parameters in a molecule superstructure and substructures to make sure that they are legitimate and reasonable. Prints error and warning messages to the display. Returns the total number of errors and, if `warnptr` is not `NULL`, the number of warnings in `warnptr`.

**structure setup**

`int molenablemols(simptr sim,int maxspecies);`
Enables molecules. This function can be called multiple times. Enter `maxspecies` as -1 for default species allocation, or to a positive number for the number of species that should be allocated. In the default, the number of species is set to 5 for the initial call, and is either left unchanged if there is spare space or doubled if there isn't space for subsequent calls. Returns 0 for success, 1 if memory could not be allocated, or 2 if `maxspecies` is less than the currently allocated number of species.

`void molsetcondition(molssptr mols,enum StructCond cond,int upgrade);`
Sets the molecule superstructure condition to `cond`, if appropriate. Set `upgrade` to 1 if this is an upgrade, to 0 if this is a downgrade, or to 2 to set the condition independent of its current value. If the condition is downgraded, this also downgrades the simulation structure condition.

`int addmollist(simptr sim,char *nm,enum MolListType mlt);`
Adds a molecule list named `nm` and of type `mlt` to the molecule superstructure, allocating it if needed. Returns the index of the list for success, -1 if memory could not be allocated, -2 if the list name has already been used, or -3 for illegal inputs (`mols` or `nm` was `NULL`).

`int molsetmaxspecies(simptr sim,int max);`
Sets the maximum number of molecular species to `max`+1, where the additional species represents empty molecules. This function is only supplied for backward compatibility, as it is now (version 2.23)

completely identical to `molenablemols`, which should be called instead. Returns 0 for success, 1 for insufficient memory, or 2 if `maxspecies` is smaller than the prior allocated number of species.

`int molsetmaxmol(simptr sim,int max);`
Sets the maximum number of molecules that the simulation is allowed to use to `max`. Enter `max` as -1 to specify that molecules should be allocated as needed without bound, which is the default behavior. This does not allocate any molecules or molecule lists. This function does not need to be called at all. This works during initial setup, or later on. Returns 0 for success, 1 if memory could not be allocated, or 5 if the requested `max` value is less than the current number of allocated molecules.

`int moladdspecies(simptr sim,char *nm);`
Adds species named `nm` to the list of species that is in the molecule superstructure. This enables molecule support if it hasn't been enabled already. Returns a positive value corresponding to the index of a successfully adds species for success, -1 for failure to allocate memory, -4 if if trying to add a species named "empty", -5 if the species already exists, or -6 if the species name includes wildcards (which are forbidden).

`int molgeneratespecies(simptr sim,const char *name,int nparents,int parent1,int parent2);`
Generates a new molecular species which has name `name`. If this molecule is generated as the product of a reaction, then enter the number of reactants in `nparents` and the reactant species numbers in `parent1` and `parent2`. Only `nparents` of these values are considered by the function. Using the parent values, this function generates diffusion coefficient and display parameters for the new species. If there is one parent, the new values are the same as those for the parent. If there are two parents, the new diffusion coefficient is $D_{new} = (D_1^{-3} + D_2^{-3})^{-1/3}$, the new display size is $r_{new} = (r_1^3 + r_2^3)^{1/3}$, and the new color is $rgb_{new} = (r_1 rgb_1 + r_2 rgb_2)/(r_1 + r_2)$, where $D$ is a diffusion coefficient, $r$ is a display size, and $rgb$ is one of the red-green-blue color values. This is the same scheme used in smolbng.c.

Returns the new species index for success, -1 for failure to allocate memory, -4 if trying to add a species named "empty", -5 if the species already exists, or -6 if the species name includes wildcards (which is forbidden here).

`void molsupdateparams(molssptr mols,double dt);`
Calculates the `difstep` parameter of the molecule superstructure and also sets the `diffuselist` set of flags in the molecule superstructure. `dt` is the simulation time step. This function should be called during initial setup (this is called from `molsupdate`), if any diffusion coefficient changes (performed with `molsetdifc`), or if any diffusion matrix changes (performed with `molsetdifm`, which also updates the diffusion coefficient).

`void molsupdatelists(simptr sim);`
Updates molecule superstructure from the level of `SClists` to `SCparams`. Can be run multiple times.

`int molsupdate(simptr sim);`
This sets up or updates the molecule superstructure. It may be called at program startup or at any later time. This sets up, or updates all molecule superstructure parameters, and works in all situations. It sets up the Gaussian table, live lists, live list lookup numbers, and diffusion step lengths. It does not process individual molecules (i.e. sorting and boxes). Returns 0 for success, or 1 for insufficient memory.

**adding and removing molecules**

`void molkill(simptr sim,moleculeptr mptr,int ll,int m);`
Kills a molecule from one of the live lists. `mptr` is a pointer to the molecule and `ll` is the list that it is currently listed in (probably equal to `mptr->list`, but not necessarily). If it is known, enter the index of the molecule in the master list (i.e. not a box list) in `m`; if it's unknown set `m` to -1. If the molecule should be killed without triggering list sorting (a rare occurrence), then send in `ll` as -1. This function resets most parameters of the molecule structure, but leaves it in the master list and in a box for later sorting by `molsort`. The appropriate `sortl` index is updated.

`moleculeptr getnextmol(molssptr mols);`
Returns a pointer to the next molecule on the dead list so that its data can be filled in and it can be added to the system. The molecule serial number is assigned. This updates the `topd` element of the molecule superstructure. Returns `NULL` if there are no more available molecules. The intention is that this function should be called anytime that molecules are to be added to the system.

`moleculeptr newestmol(molssptr mols);`
Returns a pointer to the molecule that was most recently added to the system, assuming that `molsort` has not been called in the meantime. For example, if 1 molecule is successfully added with `addmol`, `addsurfmol`, or `addcompartmol`, this will return a pointer to that molecule.

`int addmol(simptr sim,int nmol,int ident,double *poslo,double *poshi,int sort);`
Adds `nmol` molecules of type `ident` and state `MSsoln` to the system. These molecules are not added to surfaces. Their positions are chosen randomly within the rectanguloid that is defined by its corners `poslo` and `poshi`. Set these vectors equal to each other for all molecules at the same point. Set `sort` to 1 for complete sorting immediately after molecules are added and 0 for not. Returns 0 for success, 1 for out of memory, or 3 for more molecules being added than permitted with `mols->maxdlimit`.

`int addsurfmol(simptr sim,int nmol,int ident,enum MolecState ms,double *pos,panelptr pnl,int surface,enum PanelShape ps,char *pname);`
Adds `nmol` surface-bound molecules, all of type `ident` and state `ms`, to the system. They can be added to a specific panel by specifying the panel in either of two ways: send in its pointer in `pnl`, or specify the panel shape in `ps` and the panel name in `pname`. To add to all panels on the surface, send in `pnl` equal to `NULL` and/or set `ps` to `PSall`. To add the molecules to a certain point, send it in with `pos`, and otherwise set `pos` to `NULL` for random positions (there is no check that `pos` is actually on or near the panel). The function returns 0 for successful operation, 1 for inability to allocate temporary memory space, 2 for no panels match the criteria listed, or 3 for insufficient permitted molecules. See the `surfacearea` description for more information about the parameter input scheme.

For multiple panels, this function creates tables that list the cumulative areas of the included panels and the panel pointer for each included panel.

`int addcompartmol(simptr sim,int nmol,int ident,compartptr cmpt);`
Adds `nmol` molecules of type `ident` and state `MSsoln` to the system with random locations that are within compartment `cmpt`. Returns 0 for success, 2 if a random point cannot be found, or 3 if there aren't enough available molecules.

**core simulation functions**

`int molsort(simptr sim,int onlydead2live);`
Sorts molecules between live and dead lists, and between live lists. This also takes care of the live lists within boxes, as well as all list indices. Sorting is based solely on the list element of the molecule structure. Molecule ordering in lists is not preserved. If a molecule is in the system (in a master live list of type `MLTsystem`), its box element must point to a box, and those boxes' molecule lists must list the respective molecules. Resurrected molecules need to have the proper box listed in the molecule structure, but should not be listed in the box list; this listing is taken care of here. The routine returns 0 for normal operation and 1 if memory could not be allocated.

Under normal operation, the `onlydead2live` option is set to 0. In this case, the function first sets the `topl` indices to the ends of the live lists, so that future functions can see what's new in those lists. Then, it checks molecules in live lists, starting with the `sortl` molecule, which is the bottom one that is known to need sorting, and checks up to the `topl-1` molecule, which is the end of the list, as it was when this function was called. Any missorted molecules get moved. Next, the function processes all molecules in the dead list, starting from `topd` and going to `nd`, which is called the resurrected portion of the dead list. All of these molecules are moved to the appropriate live list. Afterwards, `nd` is decreased to equal `topd`, meaning that the resurrected list has zero length and that there's nothing more to do there. Finally, the function sets the `sortl` indices to equal the `nl` indices, to show that all of the

molecules in the live lists have been sorted. Thus, at the end, `topl` $\leq$ `sortl` = `nl`, where the sublist from `topl` and `nl` is the reborn list. Also, `topd` = `nd`, showing that there is no resurrected list.

On occasion, it's helpful to set the `onlydead2live` option to 1. In this case, the function only sorts resurrected molecules in the dead list into the live lists. It does not change the `sortl` or `topl` indices. One result is that any molecules that were considered to be reborn before are still considered to be reborn (and, in fact, there are now more reborn molecules due to their addition from the resurrected list). This option is helpful when molecules that have been added to the system need to be made available for other functions, such as in some commands that add molecules and for molecules added from the lattice code.

`int moldosurfdrift(simptr sim,moleculeptr mptr,double dt);`

Performs surface drift on molecule `mptr` over time step `dt`. This function should only be called if it is known that this molecule is surface-bound and that the surface drift data structure has been allocated at least down to the level of `surfdrift[i][ms]`. It should also be called before other drift or diffusion functions, because the molecule's position on the surface may affect its surface drift vector.

`int diffuse(simptr sim);`

`diffuse` does the diffusion for all molecules over one time step using single-threaded operation. Collisions with walls and surfaces are ignored and molecules are not reassigned to the boxes. If there is a diffusion matrix, it is used for anisotropic diffusion; otherwise isotropic diffusion is done, using the `difstep` parameter. The `posx` element is updated to the prior position and `pos` is updated to the new position. Surface-bound molecules are diffused as well, and they are returned to their surface. Returns 0 for success and 1 for failure (which is impossible for this function).

## 4.3 Walls (functions in smolwall.c)

The simulation volume is defined by its bounding walls. If no other surfaces are defined, these walls can be reflecting, periodic, absorbing, or transparent. Because walls can be transparent, molecules can leave the simulation volume. However, this can be a bad idea because the virtual boxes are defined to exactly fill the volume within the walls, so molecules or surfaces outside of the simulation volume can lead to very slow simulations. Also, the graphics are designed for the simulation volume within the walls. If surfaces are defined, then walls, regardless of how they are set up, are simulated as though they are transparent.

Walls are quite simple, defined with only a simple structure and no superstructure. A simulation always has 2*`dim` walls.

```
typedef struct wallstruct {
  int wdim;                  // dimension number of perpendicular to wall
  int side;                  // low side of space (0) or high side (1)
  double pos;                  // position of wall along dim axis
  char type;                   // properties of wall
  struct wallstruct *opp;        // pointer to opposite wall
  } *wallptr;
```

`wallstruct` (declared in smollib.h) is a structure used for each wall. The type may be one of four characters, representing the four possible boundary conditions.

| type | boundary |
|---|---|
| r | reflecting |
| p | periodic |
| a | absorbing |
| t | transparent |

Pointers to the opposite walls are used for wrap-around diffusion, but are simply references. There is no superstructure of walls, but, instead a list of walls is used. Walls need to be in a particular order: walls numbered 0 and 1 are the low and high position walls for the 0 coordinate, the next pair are for the 1 coordinate, and so on up to the 2*`dim`-1 wall. These walls are designed to be bounds of simulated space,

and are not configured well to act as membranes. Wall behaviors are completely ignored if any membranes are declared.

**low level utilities**

`void systemrandpos(simptr sim,double *pos);`
Returns a random point within the system volume, chosen with a uniform distribution.

`double systemvolume(simptr sim);`
Returns the total volume of the system.

`void systemcorners(simptr sim,double *poslo,double *poshi);`
Returns the low and high corners of the system volume in `poslo` and `poshi`, respectively. Both results are optional; enter `NULL` if a point is unwanted.

`void systemcenter(simptr sim,double *center);`
Returns the center of the system.

`double systemdiagonal(simptr sim);`
Returns the diagonal length of the system, or just the length if it is 1-D.

`int posinsystem(simptr sim,double *pos);`
Returns 1 if `pos` is within the system boundaries (equal to the edges counts as inside) and 0 if it is outside.

`double wallcalcdist2(simptr sim,double *pos1,double *pos2,int wpcode,double *vect);`
Calculates squared distance between point `pos1` and point `pos2`, while accounting for periodic boundaries. These are accounted for using `wpcode`, which is the wrapping code for the box that `pos1` is in. This code needs to be entered. If it's not known, then find the box pointer for the two positions with `bptr1=pos2box(sim,pos1)` and similarly for `pos2`, then set `b2` to be the index of `bptr2` within the list `bptr1->neigh`, and finally use `bptr1->wpneigh[b2]` as `wpcode`. Also, `vect` needs to be entered as a `dim`-dimensional vector of doubles. It is returned as the vector from `pos1` to `pos2`, while accounting for wrapping.

**memory management**

`wallptr wallalloc(void);`
`wallalloc` allocates and initializes a new wall. The pointer to the opposite wall needs to be set.

`void wallfree(wallptr wptr);` `wallfree` frees a wall.

`wallptr *wallsalloc(int dim);`
`wallsalloc` allocates an array of pointers to 2*`dim` walls, allocates each of the walls, and sets them to default conditions (reflecting walls at 0 and 1 on each coordinate) with correct pointers in each opp member.

`void wallsfree(wallptr *wlist,int dim);`
`wallsfree` frees an array of 2*`dim` walls, including the walls.

**data structure output**

`void walloutput(simptr sim);`
`walloutput` prints the wall structure information, including wall dimensions, positions, and types, as well as the total simulation volume.

`void writewalls(simptr sim,FILE *fptr);`
Writes all information about the walls to the file `fptr` using a format that can be read by Smoldyn. This allows a simulation state to be saved.

`int checkwallparams(simptr sim,int *warnptr);`
Checks some parameters of simulation walls to make sure that they are reasonable. Prints warning messages to the display. Returns the total number of errors and, if `warnptr` is not `NULL`, the number of warnings in `warnptr`.

**structure setup**

`int walladd(simptr sim,int d,int highside,double pos,char type);`
Adds a wall to the system. If no walls have been added yet, this allocates the necessary memory. `d` is the dimension that the wall bounds, `highside` is 0 if the wall is on the low side of the system and 1 if it is on the high side of the system, `pos` is the location of the wall in the `d` dimension, and `type` describes the boundary condition (if there aren't any surfaces). Returns 0 for success, 1 for unable to allocate memory, or 2 if the simulation structure `dim` element hasn't been set up yet.

`int wallsettype(simptr sim,int d,int highside,char type);`
Sets the type of an existing wall for dimension `d` to `type`. Set `highside` to 0 if the wall is on the low side of the system and 1 if it is on the high side of the system. Enter `d` and/or `highside` with a negative number to indicate "all" dimensions and/or system sides.

**core simulation functions**

`void checkwalls(simptr sim,int ll,int reborn,boxptr bptr);`
`checkwalls` does the reflection, wrap-around, or absorption of molecules at walls by checking the current position, relative to the wall positions (as well as a past position for absorbing walls). Only molecules in live list `ll` are checked. If `reborn` is 1, only the newly added molecules are checked; if it's 0, the full list is checked. It does not reassign the molecules to boxes or sort the live and dead ones. It does not matter if molecules are assigned to the proper boxes or not. If `bptr` is `NULL`, all diffusing molecules are checked, otherwise only those in box `bptr` are checked.

## 4.4 Reactions (functions in smolrxn.c)

Reactions were overhauled for Smoldyn version 1.82, so the following text describes the current version. Reactions are stored with several structures. There is a reaction superstructure for each reaction order (which may be 0, 1, or 2). Within each superstructure, there is a separate structure for each reaction.

### enumerated types

Following are the enumerated types and the structures.

```
#define MAXORDER 3
#define MAXPRODUCT 16
enum RevParam {RPnone,RPirrev,RPconfspread,RPbounce,RPpgem,RPpgemmax,RPpgemmaxw,
    RPratio,RPunbindrad,RPpgem2,RPpgemmax2,RPratio2,RPoffset,RPfixed};
```

The constant `MAXORDER` is one more than the maximum reaction order that is permitted. For now, order 3 and higher reactions are not supported, although much of the code should function with any reaction order. High-order reactions may be supported in future versions. `MAXPRODUCT` is the maximum number of products that a reaction can have, which is only used at present in loading reactions from a configuration file. The enumerated type `RevParam` lists the possible "reversible parameter types" that are allowed.

### reaction structure

```
typedef struct rxnstruct {
  struct rxnsuperstruct *rxnss; // pointer to superstructure
  char *rname;                  // pointer to name of reaction
  int *rctident;                // list of reactant identities [rct]
  enum MolecState *rctstate;    // list of reactant states [rct]
```

```
int *permit;                // permissions for reactant states [ms]
int nprod;                  // number of products
int *prdident;              // list of product identities [prd]
enum MolecState *prdstate;    // list of product states [prd]
long int *prdserno;         // list of product serno rules [prd]
int *prdintersurf;          // list of product intersurface rules [prd]
listptrli logserno;         // list of serial nums for logging reaction
char *logfile;              // filename for logging reaction
double rate;                // requested reaction rate
double multiplicity;          // rate multiplier
double bindrad2;            // squared binding radius, if appropriate
double prob;                // reaction probability
double chi;               // diffusion-limited fraction
double tau;                 // characteristic reaction time
enum RevParam rparamt;        // type of parameter in rpar
double rparam;              // parameter for reaction of products
double unbindrad;           // unbinding radius, if appropriate
double **prdpos;            // product position vectors [prd][d]
int disable;                // 1 if reaction is disabled
struct compartstruct *cmpt; // compartment reaction occurs in, or NULL
struct surfacestruct *srf;    // surface reaction on, or NULL
} *rxnptr;
```

Each individual reaction, of any order, is is stored in a reaction structure, `rxnstruct`. `rname` is a pointer to the reaction name that is stored in, and owned by, the reaction superstructure. `rctident` is a list of the reactant identities for the reaction, listed in the same order in which they were listed in the configuration file. Other than the order of the reactants, which is not stored elsewhere, the list of reactants that is stored here is redundant with the `table` element of the reaction superstructure. `rctstate` is a list of the allowed reactant states, again listed in the same order in which the reactants were listed in the configuration file. Each item of the `rctstate` list may be a single state, `MSnone`, `MSall`, or `MSsome`. These are fairly self-explanatory; `MSsome` means that more than one reactant state is allowed to react, but not all states. `permit`, which is largely redundant with `rctstate`, is a list of flags for which reactant states, or state combinations, are allowed to react in this reaction. State combinations can be created or interpreted with the functions `rxnpackstate` and `rxnunpackstate`. For order 2 and above, `permit` is not necessarily symmetric: for example, if solution and front are permitted, this does not imply that front and solution are permitted (however, `permit` is symmetric if multiple reactant identities are the same).

`nprod` is the number of products for the listed reaction, which may be any non-negative number. `prdident` and `prdstate`, which are arrays that are indexed from 0 to `rxn->nprod-1`, list the product identities and states; in this case, only single states are allowed (i.e. not `MSall` or `MSnone`, although `MSbsoln` is allowed).

`prdserno` is a list of serial number rules for products. The default is that this list is `NULL`, meaning not used. If it is used, then it lists a rule for each product. The rules are: -1 and -2 imply that the first and second reactant serial numbers should be used, respectively; -10 implies that the first product serial number should be used again, -11 is for the second product serial number, etc.; 0 implies that a new serial number should be used; and a positive number implies that that number should be used as the serial number.

`prdintersurf` is a list of product placement rules for intersurface reactions, meaning bimolecular reactions in which the two reactants are on two different surfaces. These are not allowed typically. However, they are allowed if this vector is allocated. Each value corresponds to a product. The rules are: 1 implies that this product should be placed on the surface, or relative to the surface, of the first reactant, and 2 implies that this product should be placed on the surface, or relative to the surface, of the second reactant. If the reaction has no products, then the vector can be allocated for a single element, which is supposed to have value 0 to indicate that intersurface reactions are allowed.

`logserno` and `logfile` are for logging individual reactions as they occur, which is done to the file named `logfile`. `logserno` is a list of serial numbers for which this reaction should be logged. If it is an empty list, then all instances of this reaction are logged.

`rate` is the reaction rate constant, measured in whichever unit system that the user is using for other aspects of the configuration file. The general rate units are $molecules * volume^{(\texttt{order}-1)}/time$. The

precise meaning of `rate` depends on the order of the reaction. The actual reaction rate is multiplied by `multiplicity`, which is here in case the same reaction arises multiple times through network generation. `bindrad2`, which only applies to order 2 and higher reactions, is the squared binding radius of the reactants. `prob` is, roughly, the reaction probability per time step. For zeroth order reactions, `prob` is the expectation number of reactions per time step in the entire simulation volume; for first order reactions, `prob` is the probability of a reactant reacting during one time step; and for second order reactions, `prob` is the probability of a reaction occurring between two reactants that have already diffused closer than their binding radius. `chi` is the diffusion-limited fraction for this reaction, meaning that it is the simulated steady-state reaction rate divided by the diffusion-limited rate that would occur with these parameters if the time step were equal to zero. `tau` is the characteristic time for the reaction. It is calculated from the other reaction parameters and, for order 2 reactions, from the initial concentrations of the molecules. The information in `tau` is completely redundant with information that is elsewhere.

`rparamt` is the type of the reversible parameter and `rparam` is the value of the reversible parameter. `unbindrad` applies to all reactions that have exactly 2 products; it is the unbinding radius of the products. `prdpos` is a list of product position displacements from the reaction position. See the description for `RxnSetRevparam`. `disable` is a flag that is 0 for normal operation and 1 if a reaction is disabled, meaning that it isn't run. One use of this is that some reactions should run in lattice space only and not in particle space, so these are disabled in particle space. `cmpt` is the compartment that a reaction occurs in, or `NULL` if it occurs everywhere. Conformational spread reactions, identified with `rparamt` equal to `RPconfspread` have reverse reaction rates that are not accounted for during rate calculations and the products are placed in the exact same places as the reactants.

Unimolecular reaction rates are surprisingly complicated. For a single reaction channel, they are simple. For multiple channels, they use the formula that is given in Andrews and Bray, 2004 for each reaction rate. Then, each probability is divided by one minus the sum of the prior probabilities to account for the fact that what's wanted is the conditional probability that a reaction happens, given that prior reactions did not happen. An alternate and possibly better method is used for surface actions, where the probability of each individual event is not stored, but instead the cumulative probability for the events is stored. Both methods are accurate.

### reaction superstructure

```
typedef struct rxnsuperstruct {
  enum StructCond condition;  // structure condition
  struct simstruct *sim;      // simulation structure
  int order;                  // order of reactions listed: 0, 1, or 2
  int maxspecies;             // maximum number of species
  int maxlist;                // copy of maximum number of molecule lists
  int *nrxn;                  // number of rxns for each reactant set [i]
  int **table;                // lookup table for reaction numbers [i][j]
  int maxrxn;                 // allocated number of reactions
  int totrxn;                 // total number of reactions listed
  char **rname;               // names of reactions [r]
  rxnptr *rxn;                // list of reactions [r]
  int *rxnmollist;            // live lists that have reactions [ll]
  } *rxnssptr;
```

The reaction superstructure, `rxnsuperstruct`, is a structure that is used for all of the reactions that are accounted for by the simulation, of a given order. Thus, there may be one for zeroth order reactions, another for first order reactions, and a third for second order reactions. Higher order reactions may be supported as well, although they are not currently. `condition` is the current condition of the superstructure and `sim` is a pointer to the simulation structure that owns this superstructure. `order` is the order of the reactions that are listed in this superstructure and `maxspecies` is simply a copy of the `maxspecies` value from the simulation structure. `maxlist` is the number of molecule lists that are assumed for, and thus contributes to the size of, `rxnmollist`.

`nrxn` is the number of reactions that are defined for a certain reactant code; conversions between reactant lists and reactant codes may be performed by the functions `rxnpackident` and `rxnunpackident`. `table` is a lookup table with which one inputs the reactant code (`[i]`) and the reaction number for that code (`[j]`,

which is 0 to `nrxn[i]-1`), and is given a reaction number; `table` is always symmetric with respect to reactant identities, which only applies to order 2 and higher reactions. Empty molecules are included in these lists, accessed with `nrxn[0]` and `table[0]`, where the former should always equal 0 and the latter should always be `NULL`. The reactions are listed next. `maxrxn` is the number of reactions of this order that have been allocated, while `totrxn` is the total number of reactions of this order that are currently defined. `rname` and `rxn`, which may be indexed from 0 to `totrxn-1`, are the list of reaction names, and the respective reactions, respectively. `rxnmollist`, which has $maxlist^{order}$ elements where maxlist is listed above, is a list of flags that indicate which molecule lists, or molecule list combinations, need to be checked to find reactions of this order.

### packed species identities

Several of the structure elements use packed values, which can be performed with `rxnpackident` and similar functions. Alternatively, they can be done directly according to the following scheme:

| item | examples | `order` = 1 | `order` = 2 |
|---|---|---|---|
| identity | `nrxn[i]`, `table[i]` | `i1` | `i1*maxspecies+i2` |
| state | `permit[ms]` | `ms1` | `ms1*MSMAX1+ms2` |
| live list | `rxnmollist[ll]` | `ll1` | `ll1*maxlist+ll2` |

See the Wildcards section of the Code Design section for an explanation of how reactions are set up and expanded.

### reaction functions

#### enumerated types

`enum RevParam rxnstring2rp(char *string);`
Converts string to enumerated `RevParam` type. This reads either single letter inputs or full word inputs. Unrecognized inputs are returned as `RPnone`.

`char *rxnrp2string(enum RevParam rp,char *string);`
Converts `RevParam` enumerated type variable `rp` to a word in `string` that can be displayed. Unrecognized inputs, as well as `RPnone`, get returned as "none".

`enum SpeciesRepresentation rxnstring2sr(char *string);`
Converts string to enumerated `Species Representation` type. This reads as many letters of the string as are given, seeing if they match or not. Unrecognized inputs are returned as `SRnone`.

`char *rxnsr2string(enum SpeciesRepresentation sr,char *string);`
Converts enumerated `SpeciesRepresentation` variable `sr` to a word in `string` that can be displayed. Unrecognized inputs, as well as `SRnone`, get returned as "none".

#### low level utilities

`int readrxnname(simptr sim,const char *rname,int *orderptr,rxnptr *rxnpt,listptrv *vlistptr,int rxntype);`
Using a reaction name in `rname`, this looks for it in one of several places, set by `rxntype` until it finds it. If `rxntype` is 1, this looks in the current list of reaction names, working with increasing reaction orders. If it is found, it returns the reaction order in `orderptr`, a pointer to the reaction in rxnpt, and the reaction number directly; if not, it returns -1. If `rxntype` is 3, this searches the list of rules to see if there is a reaction rule with the same reaction name. If so, this returns a pointer to the template reaction in that rule in `rxnpt`, creating one if necessary, the order in `orderptr`, and the rule number directly. If `rxntype` is 2, this searches the current list of reaction names, working with increasing order numbers. If this finds one or more names, all of the same order, that all start with `rname` and are followed by an underscore and then a number, then this lists those reaction pointers, cast as `void*`s, in `vlistptr`. In this case, the function returns 0; it also returns the first reaction that it found in `rxnpt`.

In all cases, this returns -1 if no reaction is found to match `rname`. This also returns -2 for failure to allocate memory (which is only possible for `rxntype` 2 or 3.

`int rxnpackident(int order,int maxident,int *ident);`
Packs a list of order identities that are listed in `ident` into a single value, which is returned. `maxident` is the maximum number of identities, from either the reaction superstructure or the simulation structure.

`void rxnunpackident(int order,int maxident,int ipack,int *ident);`
Unpacks a packed identity that is input in `ipack` to order individual identities in `ident`. `maxident` is the maximum number of identities, from the reaction superstructure or the simulation structure.

`enum MolecState rxnpackstate(int order,enum MolecState *mstate);`
Packs of list of `order` molecule states that are listed in `mstate` into a single value, which is returned.

`void rxnunpackstate(int order,enum MolecState mspack,enum MolecState *mstate);`
Unpacks a packed molecule state that is input in `mspack` to order individual states in `mstate`.

`int rxnreactantstate(rxnptr rxn,enum MolecState *mstate,int convertb2f);`
Looks through the reaction `permit` element to see if the reaction is permitted for any state or state combination. If not, it returns 0; if so, it returns 1. Also, if the reaction is permitted and if `mstate` is not `NULL`, the "simplest" permitted state is returned in `mstate`. Preference is given to `MSsoln` and `MSbsoln`, with other states investigated afterwards. If `convertb2f` is set to 1, any returned states of `MSbsoln` are converted to `MSsoln` before the function returns. This function always returns a single state in `mstate` (or `MSnone` if the reaction is not permitted at all), and never `MSall` or `MSsome`.

`int rxnallstates(rxnptr rxn);`
Returns 1 if the listed reaction is permitted for all reactant states and 0 if not.

`int findreverserxn(simptr sim,int order,int r,int *optr,int *rptr);`
Inputs the reaction defined by order `order` and reaction number `r` and looks to see if there is a reverse reaction. All molecule states for the input reaction that can react with reaction `r` are considered. If there is a direct reverse reaction, meaning the products of the input reaction (including states), are themselves able to react to form the reactants of the input reaction (with states that can produce reaction `r`), then the function returns 1 and the order and reaction number of the reverse reaction are pointed to by `optr` and `rptr`. If there is no direct reverse reaction, but the products of the input reaction are still able to react, the function returns 2 and `optr` and `rptr` point to the first listed continuation reaction. The function returns 0 if the products do not react with each other, if there are no reactants, or if there are no products. -1 is returned for illegal inputs. Either or both of `optr` and `rptr` are allowed to be sent in as `NULL` values if the respective pieces of output information are not of interest.

`int rxnisprod(simptr sim,int i,enum MolecState ms,int code);`
Determines if a molecule with identity `i` and state `ms` is the product of any reaction, of any order, returning 1 if so and 0 if not. `ms` can include `MSbsoln`. If `code` is 0, there are no additional conditions. If `code` is 1, the molecule also has to be displaced from the reaction position (i.e. either `confspread` or the unbinding radius is non-zero) in order to qualify.

**memory management**

`rxnptr rxnalloc(int order);`
Allocates and initializes a reaction structure of order `order`. The reaction has `order` reactants, a `permit` element that is allocated, no products, and most parameters are set to -1 to indicate that they have not been set up yet.

`void rxnfree(rxnptr rxn);`
Frees a reaction structure.

`rxnssptr rxnssalloc(rxnssptr rxnss,int order,int maxspecies);`
Allocates and initializes a reaction superstructure of order `order` and for `maxspecies` maximum number of species (the same value that is in the molecule superstructure). The superstructure is left with `nrxn` and `table` allocated but with no reactions. This function may be called more than once, which is useful for increasing `maxspecies`. On the first call, enter `rxnss` as `NULL` and enter it with the existing value on subsequent calls. `maxspecies` may not be decreased. Returns a pointer to the reaction superstructure on success, or `NULL` on inability to allocate memory.

`void rxnssfree(rxnssptr rxnss);`
Frees a reaction superstructure including all component reactions.

`int rxnexpandmaxspecies(simptr sim,int maxspecies);`
Expands the maxspecies value for all existing reaction superstructures, allocating memory as needed. These values should be kept synchronized with the master maxspecies in the molecule superstructure, so this is called whenever the master one changes. Returns 0 for success or, if memory could not be allocated, 1 plus the order of the superstructure where the failure occurred.

**data structure output**

`void rxnoutput(simptr sim,int order);`
Displays the contents of a reaction superstructure for order `order`, as well as all of the component reactions. It also does some other calculations, such as the probability of geminate reactions for the products and the diffusion and activation limited rate constants.

`void writereactions(simptr sim,FILE *fptr);`
Writes all information about all reactions to the file `fptr` using a format that can be read by Smoldyn. This allows a simulation state to be saved.

`int checkrxnparams(simptr sim,int *warnptr);`
Checks some parameters of reactions to make sure that they are reasonable. Prints warning messages to the display. Returns the total number of errors and, if `warnptr` is not `NULL`, the number of warnings in `warnptr`.

**parameter calculations**

`int rxnsetrate(simptr sim,int order,int r,char *erstr);`
Sets the internal reaction rate parameters for reaction `r` of order `order`. These parameters are the squared binding radius, `bindrad2`, and the reaction probability, `prob`. Zero is returned and `erstr` is unchanged if the function is successful. Possible other return codes are: 1 for a negative input reaction rate (implies that this value has not been defined yet, which is not necessarily an error; other parameters are not modified), 2 for order 1 reactions for which different reactant states would have different reaction probabilities, 3 for confspread reactions that have a different number of reactants and products, 4 for non-confspread bimolecular reactions that have non-diffusing reactants, or 5 for a reaction probability that is out of range.

For zeroth order reactions, `rxn->prob` is the expectation number of molecules that should be produced in the entire simulation volume during one time step, which is $rate \cdot dt \cdot volume$.

For first order reactions, `rxn->prob` is the conditional probability of a unimolecular reaction occurring for an individual reactant molecule during one time step, where the condition is that any previously listed possible reactions for this reactant were not chosen. First, this computes the unconditional probability. For this, if there is only one reaction possible, then the reaction probability is $prob = 1 - exp(-rate \cdot dt)$. However, other reaction channels affect the probability because, over a finite time step, the reactant may get used up before it has a chance to react in this reaction. The solution is in Andrews and Bray, 2004, eq. 14, which is $prob = (rate/sum)[1 - exp(-sum \cdot dt)]$, where $sum$ is the sum of all of the reaction channel rates. Next, this probability is conditioned as follows. Consider a reactant, A, which reacts to product $B_1$ with probability $p_1$, to product $B_1$ with probability $p_2$, to product $B_3$ with probaility $p_3$, etc. The algorithm is: the first path is taken with probability $p'_1$; if that

is not taken, then the next path is taken with probability $p'_2$; if that is not taken, then the next path is taken with probability $p'_3$, etc. Since there is no prior condition, $p_1 = p'_1$. The probability that path 2 is actually taken is $p_2 = p'_2(1 - p_1)$ because it is the probability that event 2 happens and that path 1 was not chosen. The probability that path 3 is actually taken is $p_3 = p'_3(1 - p_1)(1 - p_2)$. Thus, for example, $p'_3 = p_3/(1 - p_1)(1 - p_2)$. Here, `product` is the probability that prior paths were not taken and `prob` is the probability that a certain path is taken (e.g. $p_1$ or $p_2$).

For second order reactions, `rxn->bindrad2` is the squared binding radius of the reactants, found from `bindingradius`. In this case, the reverse parameter is accounted for in the reaction rate calculation if there is a direct reverse reaction and if it is appropriate (see the discussion of "Binding and unbinding radii," and the description for `findreverserxn`). The requested rate is doubled if the two reactants are the same due to the fact that there are half as many possible interactions in this case than if the two reactants are different. This is explained in the User's manual: "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." Smoldyn also doubles the requested reaction rate if one of the species is membrane-bound and the other is in solution. In this case, it is because the membrane occludes half of the binding volume, with the result that only half of it is accessible to binding. Thus, Smoldyn simulates the reaction with only half of the rate that would result if both species were in solution, which means that the requested rate needs to be doubled to get the correct rate.

`int rxnsetrates(simptr sim,int order,char *erstr);`

Sets internal reaction rate parameters for all reactions of order `order`. The return value of the function is -1 for correct operation. If errors occur, the reaction number where the error was encountered is returned and an error string is written to `erstr`, which should be pre-allocated to size `STRCHAR`. This function simply calls `rxnsetrate` for each reaction.

`int rxnsetproduct(simptr sim,int order,int r,char *erstr);`

Sets the initial separations for the products of reaction `r` of order `order`. This uses the `rparamt` and `rparam` elements of the reaction to do so, along with other required values such as the binding radius and parameters from any reverse reaction. The `unbindrad` and `prdpos` elements are set up here. If `rpart` is either `RPoffset` or `RPfixed`, then it is assumed that the product positions have already been set up; they are not modified again by this routine. Otherwise, they are set here, described next. This returns 0 for success or any of several error codes for errors. For each error, a message is written to `erstr`, which needs to have been pre-allocated to size `STRCHAR`. The error codes aren't listed here because they aren't used by the functions that call this one; in other words, they're irrelevant. For warnings, the function returns 0 but a warning string is written to `erstr`.

If the `rparamt` value is for `RPoffset` or `RPfixed`, then `prdpos` is not changed; otherwise it is. If there are 0 products, then everything is an error except for `RPnone`, `RPirrev`, and `RPconfspread` because nothing else makes sense. If there is 1 product, then the product position is set to the 0 vector for all `rparamt` values that make sense because there is no unbinding radius to worry about. However, most `rparamt` values don't make sense. If there are 2 products, the value depends on the `rparamt` value and on whether the reaction is reversible or not. In all cases, only the x coordinate of the `prdpos` vector is set to a non-zero value (the other coordinates are not touched at all, and nor should they be used). The following table is for reactions with 2 products.

| rparamt | rparam | unbindrad | prdpos |
|---|---|---|---|
| RPnone | - | 0 for irrev., error for rev. | 0 |
| RPirrev | - | 0 | 0 |
| RPconfspread | - | 0 | 0 |
| RPbounce | $\sigma_u$ | $\sigma_u$ | sum is $\sigma_u$ |
| | -1 | -1 | sum is $\sigma_b$ |

| | -2 | -2 (?) | 0 (?) |
|---|---|---|---|
| RPpgem | $\phi$ | $\sigma_u$ | difference is $\sigma_u$ |
| RPpgemmax | $\phi_{max}$ | $\sigma_u$ | difference is $\sigma_u$ |
| RPpgemmaxw | $\phi_{max}$ | $\sigma_u$ | difference is $\sigma_u$ |
| RPratio | $\sigma_u/\sigma_b$ | $\sigma_u$ | difference is $\sigma_u$ |
| RPunbindrad | $\sigma_u$ | $\sigma_u$ | difference is $\sigma_u$ |
| RPpgem2 | $\phi$ | $\sigma_u$ | difference is $\sigma_u$ |
| RPpgemmax2 | $\phi_{max}$ | $\sigma_u$ | difference is $\sigma_u$ |
| RPratio2 | $\sigma_u/\sigma_b$ | $\sigma_u$ | difference is $\sigma_u$ |
| RPoffset | - | set from prdpos | unchanged |
| RPfixed | - | set from prdpos | unchanged |

* For RPbounce, both product positions are positive and are set for a cumulative vector length of the unbinding radius if the rparam value is positive and to a cumulative vector length of the binding radius if rparam is negative.

`int rxnsetproducts(simptr sim,int order,char *erstr);`

Sets initial separations for products of all reactions of order order. This returns -1 for success and the reaction number for failure. Upon failure, this also returns erstr, which needs to have been preallocated with size STRCHAR, with an error message. See the discussion in the section called "Binding and unbinding radii" for more details.

`double rxncalcrate(simptr sim,int order,int r,double *pgemptr);`

Calculates the macroscopic rate constant using the microscopic parameters that are stored in the reaction data structure. All going well, these results should exactly match those that were requested initially, although this routine is useful as a check, and for situations where the microscopic values were input rather than the mass action rate constants.

For unimolecular reactions, this reverses the computations that are made in rxnsetrate; see that description for more details. ms1 is a state for which this reaction is permitted. Next, this goes through all reactions of the same reactant and state. For each, this computes the unconditional probability of the reaction being chosen, in prob. This is the product of the conditional probability of it being chosen (rxn2->prob), where the condition is that prior reactions were not chosen, and the probability that prior reactions were not chosen (product). This also adds up the unconditional probabilities in sum. Addition is appropriate here because a reaction can only happen along a single reaction channel, so these are exclusive probabilities. At the end, sum is the total probability that a reaction happens. Then, inversion of the equation, $sum = 1 - exp(-ratesum \cdot dt)$ gives the sum of all the rate constants. From this, the rate constant for this particular reaction is computed.

For bimolecular reactions that are reversible, the routine calculates rates with accounting for reversibility if the reversible parameter type of the reverse reaction is: RPpgem, RPpgemmax, RPpgemmaxw, RPratio, or RPunbindrad and not otherwise (this list was changed for version 2.56, to make it agree with the Smoldyn User's Manual). A value of -1 is returned if input parameters are illegal and a value of 0 is returned if the microscopic values for the indicated reaction are undefined (¡0). If the input reaction has a reverse reaction or a continuation reaction, and pgemptr is not input as NULL, then *pgemptr is set to the probability of geminate recombination of the products; if there is no reverse or continuation reaction, its value is set to -1.

`void rxncalctau(simptr sim,int order);`

Calculates characteristic times for all reactions of order order and stores them in the rxn->tau structure elements. These are ignored for 0th order reactions, are $1/k$ for first order reactions, and are [A][B]/[$k$([A]+[B])] for second order reactions. The actual calculated rate constant is used, not the requested ones. For second order, the current average concentrations are used, which does not capture effects from spatial localization or concentration changes. For bimolecular reactions, if multiple reactant pairs map to the same reaction, only the latter ones found are recorded. Also, all molecule states are counted, which ignores the permit reaction structure element.

**structure set up**

`void rxnsetcondition(simptr sim,int order,enum StructCond cond,int upgrade);`
Sets the reaction superstructure condition, for order `order`, to `cond`, if appropriate. Set `order` to the desired reaction order, or to -1 for all reaction orders. Set `upgrade` to 1 if this is an upgrade, to 0 if this is a downgrade, or to 2 to set the condition independent of its current value. If the condition is downgraded, this also downgrades the simulation structure condition.

`int RxnSetValue(simptr sim,char *option,rxnptr rxn,double value);`
Sets certain options of the reaction structure for reaction `rxn` to `value`. If `sim` is not `NULL`, this then downgrades the reaction condition to `SClists` to cause recomputation of molecule lists to check for reactions and also reaction simulation parameters. Returns 0 for success, 2 for unknown option, or 4 for an illegal value (e.g. a negative rate). In most cases, the value is set as requested, despite the error message.

If `option` is "rate", the `rate` element is set; if `option` is "multiplicity", the `multiplicity` element is set; if `option` is "multiplicity++", the multiplicity element is incremented; if `option` is "rateadd", the value is added to the current reaction rate (this has been largely or completely superseded by the multiplicity element); if `option` is "confspreadrad", the reaction type is made confspread and the squared binding radius is set; if `option` is "bindrad", the squared binding radius is set; if `option` is "prob", the probability is set; it `option` is "disable", the disable element is set (used for hybrid simulation in which reactions in one part of space might be disabled in other parts of space).

`int RxnSetValuePattern(simptr sim,const char *option,const char *pattern,const enum MolecState *rctstate,const enum MolecState *prdstate,double value,int oldnresults,const rxnptr template);`
Same as `RxnSetValue` except for one or more reactions that are described in `pattern` rather than with a pointer to a reaction. As before, `option` is the parameter to be set and `value` is the value it is to be set to. This also needs the reactant and product states, which are not part of the `pattern`. `oldresults` is the point of the index list where this should start setting values. This steps through all of the reactions listed in `pattern`, starting with `oldresults` and continuing to the end of the list, and calls `RxnSetValue` for each one. This function checks to see if a reaction has had its rate set before; if not, it sets the rate to `value` and if so, it adds `value` to the existing rate. Returns 0 for success, 2 for unknown option, 4 for an illegal value (e.g. a negative rate), or 5 for a reaction that is listed in the pattern but does not actually exist.

`int RxnSetRevparam(simptr sim,rxnptr rxn,enum RevParam rparamt,double rparam,int prd,double *pos,int dim);`
Sets the reversible parameter type and the appropriate reversible parameters for reaction `rxn`. This function is called by the product_placement portion of `simreadstring`. The parameter type, `rxn->paramt`, is set to `rpart`. If `rpart` requires a single value, which is stored in `rxn->rparam`, it is sent in with `rparam`. Otherwise, for fixed and offset parameter types, send in the product number that is being altered with `prd`, the vector with `pos`, and the system dimensionality with `dim`. This returns 0 for success, 1 as a warning that the reversible parameter type has been set before (except for offset and fixed types, where many different products need to be set), 2 for parameters that are out of bounds, 3 for an unrecognized `rparamt`, 4 for `prd` out of bounds, or 5 for a missing `pos` vector.

| rparamt | rparam | prd | pos |
|---|---|---|---|
| RPnone | - | - | - |
| RPirrev | - | - | - |
| RPconfspread | - | - | - |
| RPbounce | $\sigma_u$ or -1 | - | - |
| RPpgem | $\phi$ | - | - |
| RPpgemmax | $\phi_{max}$ | - | - |
| RPpgemmaxw | $\phi_{max}$ | - | - |
| RPratio | $\sigma_u/\sigma_b$ | - | - |

| | | | |
|---|---|---|---|
| RPunbindrad | $\sigma_u$ | - | - |
| RPpgem2 | $\phi$ | - | - |
| RPpgemmax2 | $\phi_{max}$ | - | - |
| RPratio2 | $\sigma_u/\sigma_b$ | - | - |
| RPoffset | - | product number | relative position |
| RPfixed | - | product number | relative position |

If `method` is `RPbounce`, then a negative number for the `parameter` indicates default bounce behavior, which is that molecules are separated by an amount that is equal to their previous overlap.

`void RxnCopyRevParam(simptr sim,rxnptr rxn,const rxnptr template);`
Copies the reverse reaction parameters from the `template` reaction to the `rxn` reaction. This doesn't do any checking. All inputs are required.

`void RxnSetPermit(simptr sim,rxnptr rxn,int order,enum MolecState *rctstate,int value);`
Sets the `permit` element of reaction `rxn`, which has order `order`, for the states that are included in `rctstate` to value `value`. `value` should be 0 to set permissions to forbidden or 1 to set permissions to permitted. Each item of `rctstate` may be an individual state or may be `MSall`. Other values are not allowed (and are not caught here). This does not affect other `permit` elements. If `order` is 2 and both reactants are the same, this automatically makes the `permit` matrix symmetric (e.g. if input states are `MSall` and `MSfront`, respectively, permissions will also be set for the pair `MSfront` and `MSall`).

`void RxnSetCmpt(rxnptr rxn,compartptr cmpt);`
Sets the `cmpt` element of the `rxn` reaction to compartment `cmpt`. This does no checking, and assigns regardless of whether `cmpt` is `NULL` or not.

`void RxnSetSurface(rxnptr rxn,surfaceptr srf);`
Sets the `srf` element of the `rxn` reaction to surface `srf`. This does no checking, and assigns regardless of whether `srf` is `NULL` or not.

`int RxnSetPrdSerno(rxnptr rxn,long int *prdserno);`
Sets the product serial number list for reaction `rxn` to the list given in `prdserno`. If the product serial number list had not been allocated previously, it is allocated here. Returns 0 for success or 1 for inability to allocate memory.

`int RxnSetIntersurfaceRules(rxnptr rxn,int *rules);`
Sets the product intersurface rules for reaction `rxn` to the list given in `rules`. If the product intersurface list had not been allocated previously, it is allocated here. Returns 0 for success or 1 for inability to allocate memory.

`int RxnSetRepresentationRules(rxnptr rxn,int order,const enum SpeciesRepresentation *rctrep,const enum SpeciesRepresentation *prdrep);`
Sets lattice versus particle representation for reaction reactants and products, for use in overlapping space hybrid simulation. Enter the reaction in `rxn`, the reaction order in `order`, and the desired representation for the reactants and products in the vectors `rctrep` and `prdrep`. Enter the first element of `rctrep` as `SRfree` to free the data structures. This function copies the contents of `rctrep` and/or `prdrep` to the arrays of the same name in the reaction structure.

`int RxnSetLog(simptr sim,char *filename,rxnptr rxn,listptrli list,int turnon);`
Sets reaction logging, which occurs upon reaction occurrence. Enter `filename` as the name of the file that reaction data should be logged to. This gets stored in `rxn->logfile`. Enter `rxn` as the reaction that should get logged, or as `NULL` for all reactions in the simulation. Enter `list` as a list of molecule serial numbers that should be logged, or as a list with only the value -1 if all molecules serial numbers should be logged. Finally, enter `turnon` as 1 to turn on logging and as 0 to turn off logging. In the latter case, the `filename` entry is ignored, the `rxn` entry is the reaction for which logging should be turned off, and `list` is the list of serial numbers for which logging should be turned off. Returns 0 for success, 1 for inability to allocate memory, or 2 as a warning that prior logfile was overwritten. If

logging is turned off for all serial numbers, this erases the stored logfile name as well, meaning that it would need to be re-entered if logging is wanted again.

`rxnptr RxnAddReaction(simptr sim,char *rname,int order,int *rctident,enum MolecState *rctstate,int nprod,int *prdident,enum MolecState *prdstate,compartptr cmpt,surfaceptr srf);`

Adds a reaction to the simulation, including all necessary memory allocation. `rname` is the name of the reaction, `order` is the order of the reaction, and `nprod` is the number of products. `rctident` and `rctstate` are vectors of size `order` that contain the reactant identities and states, respectively. Likewise, `prdident` and `prdstate` are vectors of size `nprod` that contain the product identities and states. This returns the just added reaction for success and `NULL` for inability to allocate memory. This allocates reaction superstuctures and reaction structures, and will enlarge any array, as needed. This function can also be used sequentially for reactants and products: first call it with reactants and 0 for `nprod`; next time, call it with the correct `order`, `NULL` for both reactant inputs, and the full product information.

`rxnptr RxnTestRxnExist(simptr sim,int order,const char *rname,const int *rctident,const enum MolecState *rctstate,int nprod,const int *prdident,const enum MolecState *prdstate,int exact);`

Tests if a reaction already exists. Set `exact` to 1 if this should just test to see if the reaction name is already in use. If not, this checks to see if some other reaction has the same `order` value, the same root of the reaction name, which is entered in `rname`, the same reactants as `rctident`, the same `order` reactant states as `rctstate`, the same `nprod` products as `prdident`, and the same `nprod` product states as `prdstate`. Returns the reaction if the reaction already exists and `NULL` if not.

`int RxnAddReactionPattern(simptr sim,const char *rname,const char *pattern,int oldnresults,enum MolecState *rctstate,enum MolecState *prdstate,compartptr cmpt,surfaceptr srf,int isrule,rxnptr *rxnpt);`

Adds one or more reactions to the simulation, where the reaction is defined in the `pattern` string. Enter `rname` as the name of the reaction if there is just one reaction, and as the root of the name if there are multiple reactions. The `pattern` string should have been created in the `molstring2pattern` function; it should have the reactants as space-separated words, then a newline character, then the products as space-separated words. Enter `oldnresults` as -1 if the current value of the index variable should be used to determine how many results from index have been considered already and as a number greater than or equal to zero to specify how many results from index have been considered already. Enter `rctstate` as a list of reactant states, where this list should have the same number of elements as the number of reactants in `pattern`, and enter `prdstate` as a list of product states, where this list should have the same number of elements as the number of products in `pattern`. `cmpt` and `srf` are optional; they give the compartment that the reaction will occur in and the surface that it will occur on, if the reaction should be restricted in these ways. Set `isrule` to 1 to indicate that this is a rule, meaning that any products that have not been declared beforehand will be created here, or to 0 to indicate that this is not a rule. If `rxnpt` is entered as non-`NULL` and if there is just one reaction created here, then it will be returned pointing to the new reaction. If more than one reaction is created here, it will be returned as `NULL`.

This function checks the current status of the pattern to see how much of it has been updated previously. Then, it loops over all reactions that have not been added to the simulation yet. If it encounters a reaction that already exists, it simply skips over it.

Returns 0 for success, -1 for inability to allocate memory, -2 if no wildcards were entered and one or more of the match species is unknown, -3 if a substitute species is unknown and the substitute species had no wildcards in it, -4 if the match string included more words than allowed by this function (which is 4 currently), -5 if a trial match string was too long to fit in `STRCHAR` characters (even if this wasn't actually a match), -6 if species generation failed, -11 for inability to allocate memory, -12 for missing ' ' operand, -13 for missing & operand, -15 for mismatched braces, or -20 for a destination pattern that is incompatible with the matching pattern (i.e. it has to have either 1 destination or the same number of destination options as pattern options), or -30 for failure to add the reaction.

`rxnptr RxnAddReactionCheck(simptr sim,char *rname,int order,int *rctident,enum MolecState *rctstate,int nprod,int *prdident,enum MolecState *prdstate,compartptr cmpt,surfaceptr srf,char *erstr);`
This is a simple wrapper for `RxnAddReaction`. Before it calls `RxnAddReaction` though, it checks as many of the input parameters as possible to make sure that they are reasonable. If they are not reasonable, it returns `NULL` and an error message in `erstr`, which should be allocated to size `STRCHAR`.

`int loadrxn(simptr sim,ParseFilePtr *pfpptr,char *line2,char *erstr);`
Loads a reaction structure from an already opened disk file described with `pfpptr`. If successful, it returns 0 and the reaction is added to `sim`. Otherwise it returns 1 and error information in `pfpptr`. If a reaction structure of the same order has already been set up, this function can use it and add more reactions to it. It can also allocate and set up a new structure, if needed. This need for this function has been largely superceded by functionality in `loadsim`, but this is kept for backward compatibility.

`int rxnupdateparams(simptr sim);`
Sets reaction structure parameters for the simulation time step. Return values are 0 for success, 1 for an error with setting either rates or products (and output to `stderr` with an error message), or 2 if the reaction structure was not sufficiently set up beforehand.

`int rxnupdatelists(simptr sim,int order);`
Sets the `rxnmollist` element of the reaction superstructure of order `order`. If one already exists, it is freed and then reallocated; otherwise it is just allocated. Afterwards, this function goes through all reactants of the superstructure, including their `permit` values, and registers their respective molecule lists in the `rxnmollist` array. Returns 0 for success, 1 for failure to allocate memory, 2 for a requested order that is greater than 2 (which is the highest that this function can handle), or 3 for molecules not being set up sufficiently.

`int rxnsupdate(simptr sim);`
Sets up reactions from data that have already been entered. This sets the reaction rates, sets the reaction product placements, sets the reaction `tau` values, and sets the molecule list flags. Returns 0 for success, 1 for failure to allocate memory, 2 for a Smoldyn bug, 3 for molecules not being set up sufficiently, 4 for an error with setting either rates or products (in this case, an error message is displayed to `stderr`), or 5 if the reaction structure was not sufficiently set up. This may be run at at start-up or afterwards.

**reaction parsing function**

`int rxnparsereaction(simptr sim,const char *word,char *line2,char *errstr);`
Parses reaction statement in configuration file. This code was in the "reaction" statement section of `simreadstring` function but became too long and complicated, so it got moved to its own function. It's only called by `simreadstring`.

**core simulation functions**

`int doreact(simptr sim,rxnptr rxn,moleculeptr mptr1,moleculeptr mptr2,int ll1,int m1,int ll2,int m2,double *pos,panelptr rxnpnl);`
Executes a reaction that has already been determined to have happened. `rxn` is the reaction and `mptr1` and `mptr2` are the reactants, where `mptr2` is ignored for unimolecular reactions, and both are ignored for zeroth order reactions. `ll1` is the live list of `mptr1`, `m1` is its index in the master list, `ll2` is the live list of `mptr2`, and `m2` is its index in the master list; if these don't apply (i.e. for 0th or 1st order reactions, set them to -1 and if either `m1` or `m2` is unknown, again set the value to -1. If there are multiple molecules, they need to be in the same order as they are listed in the reaction structure (which is only important for confspread reactions and for a completely consistent panel destination for reactions between two surface-bound molecules). The `pos` and `rxnpnl` inputs are only looked at for 0th order reactions; for these, they need to be a random position for the reaction to occur, and the panel if any.

Reactants are killed, but left in the live lists. Any products are created on the dead list, for transfer to the appropriate live list by the `molsort` routine. Molecules that are created are put at the reaction position, which is the average position of the reactants weighted by the inverse of their diffusion constants, plus an offset from the product definition. The cluster of products is typically rotated to a random orientation. If the displacement was set to all 0's (recommended for non-reacting products), the routine is fairly fast, putting all products at the reaction position. If the `rparamt` character is `RPfixed`, the orientation is fixed and there is no rotation. Otherwise, a non-zero displacement results in the choosing of random angles and vector rotations. If the system has more than three dimensions, only the first three are randomly oriented, while higher dimensions just add the displacement to the reaction position. The function returns 0 for successful operation and 1 if more molecules are required than were initially allocated. This function lists the correct box in the box element for each product molecule, but does not add the product molecules to the molecule list of the box.

For the bounce product placement, the general rule is that reactants 0 and 1 are copied over to the first two products and further products are placed at the reaction position. During the computation of the first product, `v1` is the vector from `mptr1` to `mptr2` and `dist`, or $d$, is computed as the length of this vector. If the distance equals zero, then this function pretends that the reactants were separated by $\sigma_b$ along the x-axis because that makes as much sense as anything else. Next, the scaling factor `x` is computed. If `rparam` is positive, meaning fixed unbinding radius, then `v1` needs to be lengthened by $\sigma_u - d$. Dividing this by $\sigma_u$ which is the cumulative length of the `prdpos` values, gives the scaling factor as $1 - d/\sigma_u$. Finally, this is divided by $d$ because vector `v1` has length $d$. If `rparam` is negative, meaning new separation is equal to prior overlap, then $\sigma_b - d$ is the prior overlap, $\sigma_b + (\sigma_b - d)$ is the new separation, and `v1` needs to be lengthened by $\sigma_b + (\sigma_b - d) - d$. Simplifying and then dividing by $\sigma_b$, which is the cumulative length of the `prdpos` values, gives the scaling factor as $2 - 2d/\sigma_b$. As before, this is finally divided by length $d$.

Reaction logging is fairly simple but is described anyhow. Nothing happens if `rxn->logserno` is `NULL`. Otherwise, `dorxnlog` is set to 0 for not logging and to 1 for logging. Then, it changes to 2 as the logging is in progress, or to -1 if logging failed due to a bad file name.

`int zeroreact(simptr sim);`
Figures out how many molecules to create for each zeroth order reaction and then tells `doreact` to create them. It returns 0 for success or 1 if not enough molecules were allocated initially.

`int unireact(simptr sim);`
Identifies and performs all unimolecular reactions. Reactions that should occur are sent to `doreact` to process them. The function returns 0 for success or 1 if not enough molecules were allocated initially.

`int morebireact(simptr sim,rxnptr rxn,moleculeptr mptr1,moleculeptr mptr2,int ll1,int m1,int ll2,enum EventType et,double *vect);`
Given a probable reaction from `bireact`, this checks for compartment or surface reactions, orders the reactants, checks for reaction permission, moves a reactant in case of periodic boundaries, increments the appropriate event counter, and calls `doreact` to perform the reaction. The return value is 0 for success (which may include no reaction) and 1 for failure. The `vect` input is only considered here, and must be non-`NULL`, if the event type is `ETrxn2wrap`; in this case, it is the vector from the current position of `mptr1` to the current position of `mptr2`.

`int bireact(simptr sim,int neigh);`
Identifies likely bimolecular reactions, sending ones that probably occur to `morebireact` for permission testing and reacting. `neigh` tells the routine whether to consider only reactions between neighboring boxes (`neigh`=1) or only reactions within a box (`neigh`=0). The former are relatively slow and so can be ignored for qualitative simulations by choosing a lower simulation accuracy value. In cases where walls are periodic, it is possible to have reactions over the system walls. The function returns 0 for success or 1 if not enough molecules were allocated initially.

## 4.5 Rules (functions in smolrule.c)

Rules were part of the reaction superstructure through version 2.46 but were then moved to their own superstructure in version 2.47. As with other portions of Smoldyn, rules are organized with a superstructure that contains information about all of the rules, and then with individual rule structures that contain information about the individual rules. The rules superstructure is only created if necessary. Each rule is a partially parsed set of instructions that tell Smoldyn what definitions to make whenever the rules are expanded, which may be before the simulation runs or on-the-fly during the simulation.

Reaction rules are typically only useful in combination with wildcard characters. See the Wildcards section of the Code Design chapter for the definitive description of the reaction rules.

### enumerated types

The only enumerated type in the rules definitions is the `RuleType`:

```
#define RTMAX 4
enum RuleType {RTreaction,RTdifc,RTdispsize,RTcolor,RTnone};
```

The constant `RTMAX` is the number of enumerated elements. The elements are the different types of rules, where each rule is used to specify a certain type of definition, which may be to create reactions, set diffusion coefficients, etc.

### rule structure

```
typedef struct rulestruct {
  struct rulesuperstruct *ruless; // pointer to superstructure
  enum RuleType ruletype;      // type of rule
  char *rulename;              // pointer to name of rule
  char *rulepattern;           // pattern for the rule
  int *ruledetails;            // list of rule states and restrictions
  double rulerate;             // rate constant
  rxnptr rulerxn;              // template reaction for reaction rules
  } *ruleptr;
```

The `ruless` element points to the superstructure that owns this rule. The `ruletype` element tells what type of rule this is. The `rulename` element is the name of the rule which, for reaction rules, is the name of the reaction as entered by the user without any suffix. The rule structure does not own the `rulename` pointers but instead they are copied over from the rule superstucture. The `rulepattern` is the pattern for the rule, including for a reaction rule. For example, a pattern for a bimolecular reaction has a single-species pattern for the first reactant, which is a species name possibly with wildcard characters, a space, a second single-species pattern, a newline character, and then space-separated product species names. The `ruledetails` element contains an array of integers. For reaction rules, these are the species states cast as integers for the reactants and products in their sequence, then the compartment number if the reaction is restricted to a specific compartment, and then the surface number if the reaction is restricted to a single surface. The `rulerate` element is, for reaction rules, the reaction rate entered by the user. The `rulerxn` element contains a reaction template for reaction rules.

The following table shows the contents of the different portions of the data structure for different types of rules.

| type | rate | `detailsi` | `detailsf` |
|---|---|---|---|
| `RTreaction` | rate constant | react. states, prod. states, compart., surf. | `NULL` |
| `RTdifc` | diff. coefficient | 0: state | `NULL` |
| `RTdifm` | 0 | 0: state | diff. matrix |
| `RTdrift` | 0 | 0: state | drift vector |
| `RTsurfdrift` | 0 | 0: state, 1: surface, 2: panel shape | drift vector |
| `RTmollist` | 0 | 0: state, 1: list number | `NULL` |
| `RTdispsize` | display size | 0: state | `NULL` |

| | | | |
|---|---|---|---|
| RTcolor | 0 | 0: state | color vector |
| RTsurfaction | 0 | 0: state, 1: surface, 2: face, 3: action | NULL |
| RTsurfrate | rate constant | 0: state, 1: surface, 2: ms1, 3: ms2, 4: new species | NULL |
| RTsurfrateint | probability | 0: state, 1: surface, 2: ms1, 3: ms2, 4: new species | NULL |

## rule superstructure

```
typedef struct rulesuperstruct {
  struct simstruct *sim;      // simulation structure
  int maxrule;                // allocated size of rule list
  int nrule;                  // actual size of rule list
  char **rulename;            // list of rule names
  ruleptr *rule;              // list of rules
  int ruleonthefly;           // for expanding rules on the fly
  } *rulessptr;
```

The rule superstructure includes a a list of rule names and a list of the actual rules, each with allocated size `maxrule` and actual size `nrule`. The `rulename` list contains strings with the rule names, which, for reactions, are simply the reaction names entered by the user, without any suffix. The `rule` list contains pointers to individual rules.

These lists are maintained by the `RuleAddRule` function. These rules do not need updating during the simulation. Instead, they are simply stored here and referred to when updating is required. The `ruleonthefly` element is initialized to 0. It is then set to 1 if on-the-fly rule generation is desired.

## rule functions

### enumerated types

`enum RuleType rulestring2rt(const char *string);`
Converts string to enumerated `RuleType` type. This reads full word inputs. Unrecognized inputs are returned as `RTnone`.

`char *rulert2string(enum RuleType rt,char *string);`
Converts enumerated `RuleType` type to string, which needs to be pre-allocated. Unrecognized rule types are converted to "none". The string is returned to simplify function cascading.

### memory management

`ruleptr rulealloc();`
Allocates a new rule structure. All pointers are initiallized to `NULL`, the ruletype is initiallized to `RTnone` and the `rulerate` element is initialized to -1. The `rulepattern` and `ruledetails` are allocated in `RuleAddRule`.

`void rulefree(ruleptr rule);`
Frees a rule structure, including its contents.

`rulessptr rulessalloc(rulessptr ruless,int maxrule);`
Allocates or expands the size of a rule superstructure. Send in the current rule superstucture in as `ruless` if there is one, or `NULL` if there isn't one. Send in the desired value for the allocated number of rules in `maxrule`. This creates a new superstructure if there wasn't one initially, allocates and initializes the rule and rulename lists, copies over any prior data, frees any prior data, and returns the rule superstructure. Returns `NULL` if out of memory.

`void rulessfree(rulessptr ruless);`
Frees a rule superstructure, including all of its contents.

### data structure output

`void ruleoutput(simptr sim);`
Outputs information about the rules, including the rule superstructure and all of the individual rules.

`void writerules(simptr sim,FILE *fptr);`
Not written yet. This will write the rules to a text file using Smoldyn input format.

`int checkruleparams(simptr sim,int *warnptr);`
This checks to make sure that rule parameters are reasonable. There are no checks at all here yet.

**structure set up**

`int RuleAddRule(simptr sim,enum RuleType type,const char *rname,const char *pattern,const enum MolecState *rctstate,const enum MolecState *prdstate,double rate,const int *detailsi,const double *detailsf);`
Adds a rule to the list of rules. Enter the rule type in `type`. For reaction rules, enter the rule name, which is the root of the reaction name and the entire reaction name if only one reaction results, in `rname`. Enter the rule pattern in `pattern`, the species or reactant state(s) in `rctstate`, any product states in `prdstate`, and the rate constant in `rate`. Enter additional integer details in `detailsi` and additional floating point details in `detailsf`.

This function stores the rule in the rule superstructure. This function creates and/or expands the rule list as needed. It then creates a new rule by copying in the rule name, copying in the rule pattern, and storing the reactant state, product state, compartment index, and surface index in the rule details, in that order. Returns 0 for success or 1 for inability to allocate memory.

The exact inputs depend on the rule type as follows. Note that these are inputs to this function, and not the storage of details in the data structure. In the data structure, the first elements of the `detailsi` vector include the species states and then the last elements are the same as the ones given here.

| type | rate | detailsi | detailsf |
|---|---|---|---|
| `RTreaction` | rate constant | 0: compartment, 1: surface | `NULL` |
| `RTdifc` | diff. coefficient | `NULL` | `NULL` |
| `RTdifm` | 0 | `NULL` | diff. matrix |
| `RTdrift` | 0 | `NULL` | drift vector |
| `RTsurfdrift` | 0 | 0: surface, 1: panel shape | drift vector |
| `RTmollist` | 0 | 0: list number | `NULL` |
| `RTdispsize` | display size | `NULL` | `NULL` |
| `RTcolor` | 0 | `NULL` | color vector |
| `RTsurfaction` | 0 | 0: surface, 1: face, 2: action | `NULL` |
| `RTsurfrate` | rate constant | 0: surface, 1: `ms1`, 2: `ms2`, 3: new species | `NULL` |
| `RTsurfrateint` | probability | 0: surface, 1: `ms1`, 2: `ms2`, 3: new species | `NULL` |

**core simulation functions**

`int RuleExpandRules(simptr sim,int iterations);`
Expands the rules by `iterations` iterations. Also, `iterations` can be set to -1 for expanding all rules until everything is up to date, -2 to set up on-the-fly expansion but not to actually do it, or -3 to do on-the-fly expansion if necessary. This goes through the rules sequentially. If there are no rules, this returns error code -41, which isn't actually an error. For each rule, it gets the rule information, including when it was last updated. It then calls `RxnAddReactionPattern` to update the index variable for the rule pattern and add any new reactions to the simulation. Next, it calls `RxnSetValuePattern` to set the reaction rates for the new reactions. This part of the function still needs work. Finally, it sees if all of the rules are up to date, terminating if so.

Returns 0 for success or the following errors which arise from `RxnAddReactionPattern`: -1 for inability to allocate memory, -2 if no wildcards were entered and one or more of the match species is unknown, -3 if a substitute species is unknown and the substitute species had no wildcards in it, -4 if the match string included more words than allowed by this function (which is 4 currently), -5 if a trial match string was too long to fit in `STRCHAR` characters (even if this wasn't actually a match), -6 if species

generation failed, -11 for inability to allocate memory, -12 for missing ‘ ’ operand, -13 for missing & operand, -15 for mismatched braces, or -20 for a destination pattern that is incompatible with the matching pattern (i.e. it has to have either 1 destination or the same number of destination options as pattern options), or -30 for failure to add the reaction. Other return code are -40 for a bug in `RxnSetValuePattern` and, of actual use, -41 for expansion requested but there are no rules to expand.

## 4.6 Surfaces (functions in smolsurf.c)

Surfaces are organized with a surface superstructure that contains not much more than just a list of surfaces and their names. Each of these surfaces, defined with a surface structure, has various properties that apply to the whole surface, such as its color on the front and back faces, how it is drawn, and how it interacts with diffusing molecules. A surface structure also includes lists of panels that comprise the surface. These panels may be rectangular, triangular, spherical, or other shapes. A single surface can contain many panels of multiple shapes.

### Surface geometry

The table below lists the types of panels and key aspects of how they are stored internally. Panel locations and sizes, plus some drawing information, are given with sets of `dim`-dimensional points, stored in the `point` element. There are `npts` points for a panel, listed below, where `npts` depends on both the panel shape and the system dimensionality. Additionally, each panel has a `dim`-dimensional `front` vector, which contains information about the direction that the panel faces. In some cases, such as for triangles, this is the normal vector to the surface and is redundant with the information in the points. In others, it contains additional information. For example, for spheres, only one element of `front` is used, and it is used to tell if the front of the panel is on the inside or outside of the sphere, which cannot be known from just the list of points. In the table below, `p` is used for point, and `f` is used for front.

Table: Properties of panels

| 1D | 2D | 3D |
|---|---|---|
| **rectangles, ps = PSrect** | | |
| `npts` = 1 | `npts` = 4 | `npts` = 8 |
| `p[0][0]` = location | `p[0][0...1]` = start | `p[0...3][0...2]` |
| | `p[1][0...1]` = end | = corners |
| | parallel to an axis | parallel to an axis |
| | front is on right | front has CCW winding |
| `f[0]` = ±1 | `f[0]` = ±1 | `f[0]` = ±1 |
| (+ for facing +0) | (+ for facing +axis) | (+ for facing +axis) |
| `f[1]` = 0 (perp. axis) | `f[1]` = perp. axis (0,1) | `f[1]` = perp. axis (0,1,2) |
| `f[2]` = undefined | `f[2]` = parallel axis | `f[2]` = axis parallel |
| | | to edge from point 0 to point 1 |
| | `p[2]` = normal for end 0 | `p[4]` = normal for 0-1 edge |
| | `p[3]` = normal for end 1 | `p[5]` = normal for 1-2 edge |
| | | `p[6]` = normal for 2-3 edge |
| | | `p[7]` = normal for 3-0 edge |
| **triangles, ps = PStri** | | |
| `npts` = 1 | `npts` = 4 | `npts` = 6 |
| `p[0][0]` = location | `p[0][0...1]` = start | `p[0...2][0...2]` |
| | `p[1][0...1]` = end | = corners |
| `f[0]` = ±1 | front is on right | front has CCW winding |
| (+1 for facing +0) | `f[0...1]` = normal vect. | `f[0...2]` = normal vect. |
| | `p[2]` = normal for end 0 | `p[3]` = normal for 0-1 edge |
| | `p[3]` = normal for end 1 | `p[4]` = normal for 1-2 edge |
| | | `p[5]` = normal for 2-0 edge |

| | | |
|---|---|---|
| **spheres, ps = PSsph** | | |
| npts = 2<br>p[0][0] = center<br>p[1][0] = radius<br><br><br>f[0] = ±1<br>(+ for front outside)<br>f[1...2] = undefined | npts = 2<br>p[0][0...1] = center<br>p[1][0] = radius<br>p[1][1] = slices<br><br><br>f[0] = ±1<br>(+ for front outside)<br>f[1...2] = undefined | npts = 2<br>p[0][0...2] = center<br>p[1][0] = radius<br>p[1][1] = slices<br>p[1][2] = stacks<br><br>f[0] = ±1<br>(+ for front outside)<br>f[1...2] = undefined |
| **cylinders, ps = PScyl** | | |
| not allowed | npts = 5<br>p[0][0...1] = start center<br>p[1][0...1] = stop center<br>p[2][0] = radius<br><br>f[0...1] = norm. right vect.<br>f[2] = ±1<br>(+ for front outside)<br><br>p[3] = normal for end 0<br>p[4] = normal for end 1 | npts = 5<br>p[0][0...2] = start center<br>p[1][0...2] = stop center<br>p[2][0] = radius<br>p[2][1] = slices<br>p[2][2] = stacks<br>f[2] = ±1<br>(+ for front outside)<br>f[0...1] = undefined<br>p[3] = normal for end 0<br>p[4] = normal for end 1 |
| **hemispheres, ps = PShemi** | | |
| not allowed | npts = 3<br>p[0][0...1] = center<br>p[1][0] = radius<br>p[1][1] = slices<br><br>p[2][0...1] = outward vect.<br>f[0] = ±1<br>(+ for front outside)<br>f[1...2] = undefined | npts = 3<br>p[0][0...2] = center<br>p[1][0] = radius<br>p[1][1] = slices<br>p[1][2] = stacks<br>p[2][0...2] = outward vect.<br>f[0] = ±1<br>(+ for front outside)<br>f[1...2] = undefined |
| **disks, ps = PSdisk** | | |
| not allowed | npts = 2<br>p[0][0...1] = center<br>p[1][0] = radius<br><br>f[0...1] = normal vect.<br>f[2] = undefined | npts = 2<br>p[0][0...2] = center<br>p[1][0] = radius<br>p[1][1] = slices<br>f[0...2] = normal vect. |

To add a new panel shape, several things need to be done. Add the panel shape name to `PanelShape` and increment the `#define` constant `PSMAX`, which are defined in the smoldyn.h header file. Define the panel `point` and `front` values in the table above. Add the new panel shape to the following functions (and maybe others): `psstring2ps`, `ps2psstring`, `panelpoints`, `loadsurface` (panel input section), `surfaceoutput`, `panelside`, `surfacearea`, `panelrandpos`, `lineXpanel`, `fixpt2panel`, `surfacereflect`, `surfacejump`, and `panelinbox`. Also, add the panel shape to `RenderSurfaces` in smoldyn.c. Most of these are relatively easy, although some math likely needs to be done for a couple of them. Finally, check and document.

## Molecule-surface interactions

Molecule-surface interactions arise when a molecule collides with a surface, or when a surface-bound molecule undergoes a spontaneous state change, such as desorption. Collisions can arise both for solution-phase molecules or surface-bound molecules; in the latter case, a molecule bound to surface A diffuses along that surface and then collides with surface B, which intersects surface A.

Surface interactions can be certain or probabilistic. The former interactions, which the user enters with the `action` configuration file statement, always happen immediately upon interaction. These certain interactions are: reflect, transmit, absorb, jump, and port. Probabilistic interactions occur with certain probabilities either upon interaction or at each time step. The user enters the rates of these interactions with the `rate` statement. This statement is also used for spontaneous transition rates of surface-bound molecules.

Internally, both the `action` and `rate` elements of the surface data structure refer to both collision interactions and spontaneous state changes of surface-bound molecules. The "face" index of these elements is either `PFfront` or `PFback` for collisions, or is `PFnone` for surface-bound state changes.

There are several ways of describing surface interactions. One can use a verb, such as reflect, transmit, or adsorb, or one can list the beginning molecule state, the surface interaction face, and the ending molecule state. Or, one can just list the beginning and ending molecules states, with pseudo-states, plus a third state when necessary (see the `surfsetrate` function and the `rate` configuration file statement). Smoldyn uses all of these methods which means that interconversions become necessary. The following table lists the states used in the action details data structure and their meanings. Conversions are given in later tables.

| interaction class | forward states | | | action |
|---|---|---|---|---|
| | `ms1` | `face1` | `ms2` | |
| collision from solution state | soln | front | fsoln | reflect |
| | " | " | bsoln | transmit |
| | " | " | bound | adsorb |
| | " | back | fsoln | transmit |
| | " | " | bsoln | reflect |
| | " | " | bound | adsorb |
| impossible | " | none | any | |
| collision from bound state | bound | front | fsoln | reflect |
| | " | " | bsoln | transmit |
| | " | " | bound | hop |
| | " | " | bound' | hop |
| | " | back | fsoln | transmit |
| | " | " | bsoln | reflect |
| | " | " | bound | hop |
| | " | " | bound' | hop |
| action from bound state | " | none | fsoln | desorb |
| | " | " | bsoln | desorb |
| | " | " | bound | no |
| | " | " | bound' | flip |

For the most part, surface-bound molecules cannot be absorbed, jumped, or ported, using the same surface. The exception is that if a surface-bound molecule in its `MSfront` or `MSback` state diffuses onto a new surface panel, and the new panel has jump behavior for its `MSsoln` or `MSbsoln` states, then the molecule is jumped.

If a surface-bound molecule collides with another surface, then the result is typically dictated by the "action" statement and data structure contents, as described above. However, if the panel that it collides with has been declared to be a neighbor of the panel that the molecule is on, then the molecule does not behave according to the action given above. Instead, it can "hop", meaning that it has a 50% chance of moving to the new panel and 50% chance of staying on its current panel, or it can "stay", meaning that it stays on its current panel. These options are stored in the surface `neighhop` data structure element, where 1 indicates hopping and 0 indicates staying. Staying is the default. The hopping option is provided so that molecules can diffuse between neighboring surface panels, even if the panels don't meet at their edges (e.g. a collection of spheres).

Note that absorption, jumping, and porting will have time-step dependent behaviors; from Andrews, *Phys. Biol.*, 2009, the absorption/ jumping/ porting coefficient is about $0.86s/\Delta t$, where $s$ is the rms step length and $\Delta t$ is the time step.

### Surface data structures

```
#define PSMAX 6                                  // maximum number of panel shapes
enum PanelFace {PFfront,PFback,PFnone,PFboth};
enum PanelShape {PSrect,PStri,PSsph,PScyl,PShemi,PSdisk,PSall,PSnone};
enum SrfAction {SAreflect,SAtrans,SAabsorb,SAjump,SAport,SAmult,SAno,SAnone,
    SAadsorb,SArevdes,SAirrevdes,SAflip};
enum DrawMode {DMno=0,DMvert=1,DMedge=2,DMve=3,DMface=4,DMvf=5,DMef=6,DMvef=7,
    DMnone};
enum SMLflag {SMLno=0,SMLdiffuse=1,SMLreact=2,SMLsrfbound=4};
```

Panel faces can be front or back, and there are also enumerations for both and none. For version 2.19, I changed the enumeration sequence to put `PFnone` before `PFboth`. This sequence is important because the surface `action` and `rate` elements are allocated for the first three enumerated panel faces, but not more.

Panel shapes are enumerated with `PanelShape`, of which there are `PSMAX` shapes, plus enumerations for all and none, which can be useful as function arguments.

Not all surface actions apply to all circumstances. For example, reflect, transmit, absorb, and jump only apply to collisions between diffusing molecules and surfaces (plus, diffusing molecules in front and back states can jump as well, although the jump action is not assigned to these states). "no" applies to surface-bound molecules, meaning that they are static and don't change over time; in contrast, those that might change are labeled as `SAmult`, meaning that there are multiple possible outcomes at each time step. "none" does not imply "no action", but means instead "none of the other options". "port" is an action for exporting molecules to other simulators, such as MOOSE. The actions `SAadsorb`, `SArevdes` (reversible desorption), `SAirrevdes` (irreversible desorption), and `SAflip` (change of surface-bound state), are returned by the `surfaction` function, but are not options that the user can choose. This is because they are, in a sense, sub-actions within the `SAmult` option and they cannot be chosen for exclusive use.

`DrawMode` lists the drawing modes for polygons or other surfaces. `vert` or `v` are for vertices, `edge` or `e` is for edges, and `face` or `f` are for faces. Where multiple options are listed, Smoldyn is supposed to draw multiple methods simultaneously, although I don't believe that it supports this at present. Numbers are explicitly specified in `DrawMode` because there are distinct bits for vertex, edge, or face, which allows them to be extracted from the code using bitwise logic operations.

`SMLflag` (stands for surface molecule list) lists binary flags for the molecule lists for which surface checking is required. For example, if the `SMLdiffuse` flag is set for some molecule list, then surfaces need to be checked for all molecules in that list after diffusion occured. Similarly, those with `SMLreact` set need to be checked after reactions occur, because they might have been placed across a surface during product placement. Those with `SMLsrfbound` set are surface bound molecules that might be able to desorb or flip orientation.

```
typedef struct surfactionstruct {
  int *srfnewspec;              // surface convert mol. species [ms]
  double *srfrate;              // surface action rate [ms]
  double *srfprob;              // surface action probability [ms]
  double *srfcumprob;           // surface cumulative probability [ms]
  int *srfdatasrc;              // surface data source [ms]
  double *srfrevprob;           // probability of reverse action [ms]
  } *surfactionptr;
```

The surface action structure collects together details for surface actions. It is generally only allocated if a surface action for a specific molecular species, molecular state, and surface interaction face is of type `SAmult`, meaning that multiple possible outcomes are possible. In that case, it records the rate at which conversion can take place to each of the possible output states, the probability of each of these transitions for each time step, the cumulative probabilities (used for efficient simulation), the new species that should be created upon transition (usually the same as the current species, but not necessarily), and the source of the interaction rate data. Each of the vectors in the surface action structure is allocated to size `MSMAX1`, meaning that `MSbsoln` is an allowed outcome state. The `srfdatasrc` value is initialized to 0, is set to 1 if the user entered an interaction rate, or is set to 2 if the user entered an interaction probability.

```
typedef struct panelstruct {
  char *pname;                  // panel name (reference, not owned)
```

```
  enum PanelShape ps;          // panel shape
  struct surfacestruct *srf;     // surface that owns this panel
  int npts;                  // number of defining points
  double **point;              // defining points, [number][d]
  double **oldpoint;               // prior defining points, [number][d]
  double front[DIMMAX];        // front parameters, which depend on the shape
  double oldfront[DIMMAX];         // prior front parameters, which depend on the
      shape
  struct panelstruct *jumpp[2];// panel to jump to, if appropriate [face]
  enum PanelFace jumpf[2];       // face to jump to, if appropriate [face]
  int maxneigh;                // maximum number of neighbor panels
  int nneigh;                  // number of neighbor panels
  struct panelstruct **neigh; // list of neighbor panels [p]
  double *emitterabsorb[2];    // absorption for emitters [face][i]
  } *panelptr;
```

`pname` is a pointer to the panel name, which is contained in the surface structure; this memory is owned by the surface, not by the panel. `ps` is the panel shape. `srf` is a pointer to the surface that owns this panel; it would be called a `surfaceptr`, except that a `surfaceptr` isn't declared until later. `npts` is the number of `dim`-dimensional points that are allocated for this panel. `point` and `front` have meanings that depend on the panel shape and on the dimensionality, described in the preceding table. `jumpp` and `jumpf` are used for periodic boundary conditions and jumping molecules; these are the panel and face that a molecule will be sent to if it collides with the front or back face of this panel. `maxneigh` is the number of neighbor references that are allocated, `nneigh` is the number of neighboring panels and `neigh` is the list of neighboring panels. These neighboring panels may be within the same surface or on a different surface. These are used for diffusion of surface-bound molecules. Neighbors are only allocated as necessary (by the `surfsetneighbors` function). `emitterabsorb[face][i]` is the panel absorption probability for face `face` and species `i` to account for emitters (see the user's manual). It is only allocated if necessary.

The `oldpoint` and `oldfront` elements are for moving surfaces. They are used to show where a surface came from. They are used only when a surface is about to be moved and during the moving process.

```
typedef struct surfacestruct {
  char *sname;                 // surface name (reference, not owned)
  struct surfacesuperstruct *srfss; // owning surface superstructure
  int selfindex;               // index of self
  enum SrfAction ***action;    // action for molecules [i][ms][face]
  surfactionptr ***actdetails;  // action details [i][ms][face]
  int neighhop;                // whether molecules hop between neighbors
  double fcolor[4];            // RGBA color vector for front
  double bcolor[4];            // RGBA color vector for back
  double edgepts;              // thickness of edge for drawing
  unsigned int edgestipple[2];  // edge stippling [factor,pattern]
  enum DrawMode fdrawmode;     // polygon drawing mode for front
  enum DrawMode bdrawmode;     // polygon drawing mode for back
  double fshiny;               // front shininess
  double bshiny;               // back shininess
  int maxpanel[PSMAX];         // allocated number of panels [ps]
  int npanel[PSMAX];           // actual number of panels [ps]
  char **pname[PSMAX];         // names of panels [ps][p]
  panelptr *panels[PSMAX];     // list of panels [ps][p]
  struct portstruct *port[2]; // port, if any, for each face [face]
  double totarea;              // total surface area
  int totpanel;                // total number of panels
  double *areatable;           // cumulative panel areas [pindex]
  panelptr *paneltable;        // sequential list of panels [pindex]
  int *maxemitter[2];          // maximum number of emitters [face][i]
  int *nemitter[2];            // number of emitters [face][i]
  double **emitteramount[2];  // emitter amounts [face][i][emit]
  double ***emitterpos[2];     // emitter positions [face][i][emit][d]
```

```
  int *maxmol;                  // allocated size of live lists [ll]
  int *nmol;                    // number of molecules in live lists [ll]
  moleculeptr **mol;            // live molecules on the surface [ll][m]
   } *surfaceptr;
```

selfindex is the index of the surface in the surface structure; this is useful in case the surface is sent to a function using a pointer and the function needs to know which surface it is. action lists the the actions that happen to molecules for each state. It is allocated to size [maxspecies][MSMAX][3], which means that it does not allow the MSbsoln state and also that it allows PFfront, PFback, or PFnone panel faces. An action may be, for example, SAreflect or SAno, if these actions always happen and is SAmult if there are multiple possible actions that could happen at any particular time step. This vector has maxident elements times MSMAX elements to account for each molecule species and then each state. Only for those that have type SAmult, is the data pointed to by actdetails relevant. As described above, these details list the action rates, probabilities, and other values for transitions between states for each molecular species. neighhop tells whether surface-bound molecules should be allowed to hop to neighboring panels upon collision or whether they should stay on the originating panel.

fcolor and bcolor are the colors of the front and back of the surface in the order: red, green, blue, alpha; each has a value between 0 and 1. edgepts is the thickness of edges in points for drawing, which applies to all drawing situations except for 3D and when the surface faces are rendered. fdrawmode and bdrawmode describe how the surface front and back should be drawn. Not all options apply to 1D and 2D simulations. fshiny and bshiny are shininess values for OpenGL surface rendering. maxpanel and npanel are the number of panels that are allocated or used, respectively, for each of the panel shapes. panelname lists a name for each panel. panels are lists of pointers to the panels for the possible shapes. Note that every panel within a surface has the same drawing scheme and the same interaction with molecules. port points to the port structure that applies to each face, if any.

Considering all surface panels, the total surface area is totarea and there are totpanel panels. These panels are listed sequentially in the list paneltable, and their cumulative areas are listed in areatable. Thus, for example, areatable[0] is the area of the first panel and areatable[totpanel-1] is equal to totarea. These four elements are set up in surfacesupdate.

Although it is a rather specialized function, surfaces can be configured to absorb molecules with coefficients that yield concentrations that are the same as those for unbounded systems (see the user's manual). This configuration relies on the definitions of point "emitters". For face face and species i, maxemitter[face][i] emitters are allocated, of which nemitter[face][i] are actually used. These emitters have amount emitteramount[face][i][emit] (emit is the emitter index) and are at locations emitterpos[face][i][emit][d], where d is the index for the dimensionality.

For Smoldyn version 2.50, I added molecule lists in surfaces, so surfaces know what molecules are adsorbed to them. These lists are in the maxmol, nmol, and mol lists. Molecules are assigned to these lists in the assignmolecs function in smolboxes.c.

```
typedef struct surfacesuperstruct {
  enum StructCond condition;  // structure condition
  struct simstruct *sim;      // simulation structure
  int maxspecies;             // maximum number of molecular species
  int maxsrf;                 // maximum number of surfaces
  int nsrf;                   // number of surfaces
  double epsilon;             // max deviation of surface-point from surface
  double margin;              // panel margin away from edge
  double neighdist;           // neighbor distance value
  char **snames;              // surface names [s]
  surfaceptr *srflist;        // list of surfaces [s]
  int maxmollist;             // number of molecule lists allocated
  int nmollist;               // number of molecule lists used
  enum SMLflag *srfmollist;   // flags for molecule lists to check [ll]
  } *surfacessptr;
```

This is the superstructure for surfaces. condition is the current condition of the superstructure and sim is a pointer to the simulation structure that owns this superstructure. maxspecies is a copy of maxspecies

from the molecule superstructure, and is the allocated size of the surface action, rate, and probability elements. `maxsrf` and `nsrf` are the number of surfaces that are allocated and defined, respectively. `epsilon` is a distance value that is used when fixing molecules to panels; if a molecule is already within `epsilon` of a panel and on the correct side, no additional moving is done. `margin` is the distance inside the edge of a panel to which molecules are moved if they need to be moved onto panels. `neighdist` is used for the diffusion of molecules on a surface; if the point where a molecule diffuses off of one panel is within $\texttt{neighdist}^{1/2}$ of the closest point on another panel, then the molecule can move to the neighboring panel. `snames` is a list of names for the surfaces. `srflist` is the list of pointers to surfaces. `srfmollist` is a list of flags for which molecule lists need to be checked for surface interactions; `maxmollist` and `nmollist` are local copies of `sim->mols->maxlist` and `sim->mols->nlist`, and are used to read the `srfmollist` element. The `SMLflag` enumerated values are or-ed together in these elements.

It was surprisingly difficult to get surfaces to work well enough that diffusing molecules did not leak through reflective panels. Because of that, the code is written unusually carefully, and in ways that are not necessarily obvious, so be careful when modifying it. For example, round-off error differences between two different but mathematically identical ways of calculating a molecule distance from a surface can easily place the molecule on the wrong side of a surface panel.

If a molecule is exactly at a panel, it is considered to be at the back side of the panel. Initially, I defined direct collisions as collisions in which the straight line between two points crosses a surface, whereas an indirect collision is one in which the straight line does not cross a surface but it was determined with a random number that the Brownian motion trajectory did contact the surface. Indirect collisions proved to slow down the program significantly, greatly complicate the code development, and provided minimal accuracy improvements, so I got rid of them. Now, only direct collisions are detected and dealt with.

## Function interdependence

Following is a partial listing of what functions call what other functions. This is incomplete but could be completed relatively easily if necessary.

`surfreadstring`
- name
  - `surfaddsurface`
    - `surfenablesurfaces` if needed
      - `surfacessalloc`
        - `surfacealloc`
          - `emittersalloc`
- action
  - `surfsetaction`
    - `surfsetcondition` to SCparams
- rate or rate_internal
  - `surfsetrate`
    - `surfaceactionalloc` if needed
    - `surfsetcondition` to SCparams
- color
  - `surfsetcolor`
- thickness
  - `surfsetedgepts`
- stipple
  - `surfsetstipple`
- polygon
  - `surfsetdrawmode`
- shininess
  - `surfsetshiny`
- max_panels (deprecated function)
  - `surfsetmaxpanel`
    - `panelsalloc`

panel
`surfaddpanel`
`panelsalloc` if needed
`emittersalloc`
`surfsetcondition` to SClists
`boxsetcondition` to SCparams
jump
`surfsetjumppanel`
neighbors
`surfsetneighbors`, allocates space as needed
unbounded_emitter
`surfaddemitter`
`emittersalloc` if needed
`surfsetcondition` to SCparams

`surfsetepsilon`
`surfenablesurfaces` if needed, see above

`surfsetmargin`
`surfenablesurfaces` if needed, see above

`surfsetneighdist`
`surfenablesurfaces` if needed, see above

## Surface functions

### enumerated types

`enum PanelFace surfstring2face(char *string);`
Converts panel face `string` to an enumerated panel face type. Input strings are "front" for the front, "back" for the back, or "all" or "both" for both sides. Also, initial portions of these strings, such as "f" or "fro" are sufficient. Other inputs result in `PFnone`.

`char *surfface2string(enum PanelFace face,char *string);`
Converts enumerated panel `face` to a string, in `string`, which must be pre-allocated. Output strings are "front", "back", "both", or "none". `string` is returned to allow for function nesting.

`enum SrfAction surfstring2act(char *string);`
Converts action `string` to an enumerated action type. Input strings are the same as the `SrfAction` strings. Initial portions of strings are sufficient. Unknown strings result in `SAnone`. This cannot return the actions `SAadsorb`, `SAirrevdes`, `SArevdes`, or `SAflip`, because the user cannot enter them. They are "sub-actions" of `SAmult`.

`char *surfact2string(enum SrfAction act,char *string);`
Converts enumerated surface action `act` to a string, in `string`, which must be pre-allocated. Output strings are "reflect", "transmit", etc. `string` is returned to allow for function nesting.

`enum PanelShape surfstring2ps(char *string);`
Converts panel shape `string` to an enumerated panel shape type. Input strings are the same as `PanelShape` strings.

`char *surfps2string(enum PanelShape ps,char *string);`
Converts enumerated panel shape to a string. Output strings are abbreviated shape names, such as "rect" to designate a rectangle. Also, `PSall` and `PSnone` result in the strings "all" and "none". `string` is returned to allow for function nesting.

`enum DrawMode surfstring2dm(char *string);`
Converts drawing mode `string` to an enumerated drawing mode type. Input strings are `DrawMode` names. Unrecognized input results in `DMnone`.

`char *surfdm2string(enum DrawMode dm,char *string);`
Converts enumerated drawing mode to a string. Output strings are abbreviated drawing mode names. `string` is returned to allow for function nesting.

**low level utilities**

`int readsurfacename(simptr sim,char *str,enum PanelShape *psptr,int *pptr);`
Reads the first word of string `str` to parse the surface name and an optional panel name, which are entered in the format surface:panel. Returns the surface index directly and, if the pointers are not `NULL`, returns the panel shape in `psptr` and the panel index in `pptr`. Returns the surface index, or -1 if `str` is missing, -2 if no surfaces have been defined in the current simulation, -3 if the name string cannot be read, -4 if the surface name is unknown, or -5 if the surface is "all". In `psptr` and `pptr` are returned, respectively: `PSnone` and -1 if the panel is not given, `PSnone` and -2 if the surface is "all" and the panel is something else (this is an error), `PSnone` and -3 if the panel name is unknown, `PSall` and -5 if the panel is "all", and otherwise the panel shape and panel number. These outputs were changed 4/24/12, while developing Smoldyn 2.27.

`int panelpoints(enum PanelShape ps,int dim);`
Returns the number of point elements that need to be allocated for a panel of shape `ps` and for system dimensionality `dim`. These numbers are the same as those listed in the table above. 0 is returned for inputs that don't make sense (e.g. `PSall`) or for shapes that are not permitted in the requested dimension.

`int surfpanelparams(enum PanelShape ps,int dim);`
Returns the number of numerical parameters that the user needs to enter to define a panel of shape `ps` and in a `dim` dimensional system. 0 is returned for inputs that don't make sense of for shapes that are not permitted in the requested dimension.

`void panelmiddle(panelptr pnl,double *middle,int dim,int onpanel);`
Returns the middle of panel `pnl` in the `dim`-dimensional vector `middle`; `dim` is the system dimensionality. For spheres, hemispheres, and cylinders, the middle point is the actual center location if `onpanel` is 0, which is enclosed by the panel but not on it; for these, set `onpanel` to 1 for `middle` to be returned as a point on the panel, although it will no longer be in the middle. `onpanel` is ignored for rectangles, triangles, and disks. If `onpanel` is 1: for spheres, the returned point is directly to the positive $x$ direction from the sphere center; for cylinders, the returned point is as close as possible to the center point; and for hemispheres, the returned point is the center of the on-panel locations.

`double panelarea(panelptr pnl,int dim);`
Returns the area of panel `pnl`; `dim` is the system dimensionality, as always.

`double surfacearea(surfaceptr srf,int dim,int *totpanelptr);`
Returns the total area of surface `srf`; `dim` is the system dimensionality. If `totpanelptr` is not `NULL`, it is returned with the total number of panels in the surface. This function calculates the area, rather than just returning the value that is in the `totarea` surface element.

`double surfacearea2(simptr sim,int surface,enum PanelShape ps,char *pname,int *totpanelptr);`
Returns the area of one or more panels. For the area of a single panel, the inputs are the surface number, the panel shape, and the panel name. The area is returned (number of points for 1-D, line length for 2-D, and area for 3-D). If `totpanelptr` is sent in as not `NULL`, it will point to the integer 1 on return. For multiple panels, set any or all of the inputs to "all" using: a negative number for surface, `PSall` for `ps`, and/or "all" for `pname`; `totpanelptr` will point to the number of panels included in the sum. For example, if surface is a positive number, `ps` is `PSall`, and panel is "all", then the total

area of all panels of the specified surface is found. Or, if surface is negative, `ps` is `PSall` and `pname` is "endcap" then the area is found for all panels named "endcap", regardless of their surface or shape. If no panels match the input description, 0 is returned and `totpanelptr` is left pointing to a 0.

`void panelrandpos(panelptr pnl,double *pos,int dim);`
Returns a random position, in `pos`, on the surface of panel `pnl`, in a `dim` dimensional system. The result might be on either side of the panel.

`panelptr surfrandpos(surfaceptr srf,double *pos,int dim);`
Returns a random position, in `pos`, on the surface `srf`, in a `dim` dimensional system. The result might be on either side of the surface. The return value is a pointer to the panel that the point is in, or `NULL` if the surface has no panels.

`int issurfprod(simptr sim,int i,enum MolecState ms);`
Determines if molecule identity `i` and state `ms` is the product of a surface action, accounting for all surfaces. Returns 1 if so and 0 if not. `ms` can be `MSbsoln`. This should work after surfaces have been loaded and either before or after they have been set up. This does not return 1 if molecule `i` and state `ms` only participates in surface interactions, but is not produced by them. For example, if a molecule type simply reflects off of a surface with no species or state change, then this doesn't count. On the other hand, if a molecule type is produced when a molecule of a different state adsorbs to a surface, then this does count.

`int srfsamestate(enum MolecState ms1,enum PanelFace face1,enum MolecState ms2,enum MolecState *ms3ptr);`
Determines if a molecule in state `ms2` is in the same state as it was in `ms1`, and returns 1 if so and 0 if not. Also, if `ms3ptr` is not `NULL`, this returns the state that is the same as the `ms1` and `face1` information in the value pointed to by `ms3ptr`. This returns values from the following table.

| interaction class | forward states | | | action | return values | |
|---|---|---|---|---|---|---|
| | `ms1` | `face1` | `ms2` | | function | `*ms3ptr` |
| collision from solution state | soln | front | fsoln | reflect | 1 | fsoln |
| | " | " | bsoln | transmit | 0 | fsoln |
| | " | " | bound | adsorb | 0 | fsoln |
| | " | back | fsoln | transmit | 0 | bsoln |
| | " | " | bsoln | reflect | 1 | bsoln |
| | " | " | bound | adsorb | 0 | bsoln |
| impossible | " | none | any | | 0 | none |
| collision from bound state | bound | front | fsoln | reflect | 1 | fsoln |
| | " | " | bsoln | transmit | 0 | fsoln |
| | " | " | bound' | hop | 0 | fsoln |
| | " | back | fsoln | transmit | 0 | bsoln |
| | " | " | bsoln | reflect | 1 | bsoln |
| | " | " | bound' | hop | 0 | bsoln |
| action from bound state | " | none | fsoln | desorb | 0 | bound |
| | " | " | bsoln | desorb | 0 | bound |
| | " | " | bound | no | 1 | bound |
| | " | " | bound' | flip | 0 | bound |

`void srfreverseaction(enum MolecState ms1,enum PanelFace face1,enum MolecState ms2,enum MolecState *ms3ptr,enum PanelFace *face2ptr,enum MolecState *ms4ptr);`
This function simply takes in a surface interaction are returns what the reverse interaction would be. It does not consider the specifics of individual surfaces, any interaction rates, or any other details. Instead, it simply inverts the table of surface interactions that is presented above. More specifically, given that some molecule starts with state `ms1`, interacts with `face1` of a surface, and then ends in state `ms2`, this finds the reverse surface action. In this reverse action, the molecule starts in state

ms3, interacts with face2, and ends in state ms4. These latter parameters are pointed to by ms3ptr, face2ptr, and ms4ptr, respectively. In concept, the reverse action is that ms3 should equal ms2 and ms4 should equal ms1, although it's rarely this simple. The reason is that the starting states cannot include MSbsoln, whereas the end states can include it, and also the actions have to allow for surface-bound molecules to interact with other surfaces that they cross. The following table shows the input and output values.

| interaction class | forward states | | | action | reverse states | | |
|---|---|---|---|---|---|---|---|
| | ms1 | face1 | ms2 | | ms3 | face2 | ms4 |
| | soln | front | fsoln | reflect | soln | front | fsoln |
| | ” | ” | bsoln | transmit | soln | back | fsoln |
| collision from | ” | ” | bound | bind | bound | none | fsoln |
| solution state | ” | back | fsoln | transmit | soln | front | bsoln |
| | ” | ” | bsoln | reflect | soln | back | bsoln |
| | ” | ” | bound | bind | bound | none | bsoln |
| impossible | ” | none | any | impossible | none | none | none |
| | bound | front | fsoln | reflect | bound | front | fsoln |
| | ” | ” | bsoln | transmit | bound | back | fsoln |
| collision from | ” | ” | bound’ | hop | *bound’* | *both* | *bound* |
| bound state | ” | back | fsoln | transmit | bound | front | bsoln |
| | ” | ” | bsoln | reflect | bound | back | bsoln |
| | ” | ” | bound’ | hop | *bound’* | *both* | *bound* |
| action from | ” | none | fsoln | desorb | soln | front | bound |
| bound state | ” | ” | bsoln | desorb | soln | back | bound |
| | ” | ” | bound’ | flip | bound’ | none | bound |

The italicized rows for the “reverse states” columns indicate that a bound-state molecule collided with a new surface and then hopped to this new surface. In this case, it's easy to know the beginning and ending states, but it is impossible to know the face of the reverse action. Also, the kinetics of the reverse process cannot be ascertained without knowing which surface the molecule was initially bound to. For this reason, a face2 return value of PFboth indicates that the molecule hopped from one surface to another, which means that the interaction face of the reverse action cannot be determined without further information. This might be interpreted as an error condition. Finally, if ms1 equals MSsoln and face1 equals PFnone, this is an error; in this case, the function returns “none” for all variables.

`void srftristate2index(enum MolecState ms,enum MolecState ms1,enum MolecState ms2,enum MolecState *ms3ptr,enum PanelFace *faceptr,enum MolecState *ms4ptr);`

Converts between the format that the rate configuration file statement and the surfsetrate function input, called tristate format, and the index format that the action details data structure uses. This function converts according to the following table. Rows that are listed in italics may be forbidden as input combinations, but still make logical sense and so they are converted here.

| interaction class | tristate format | | | action | index format | | |
|---|---|---|---|---|---|---|---|
| | ms | ms1 | ms2 | | ms3 | face | ms4 |
| | soln/none | soln | soln | *reflect* | soln | front | fsoln |
| | ” | ” | bsoln | transmit | soln | front | bsoln |
| collision from | ” | ” | bound | adsorb | soln | front | bound |
| solution state | ” | bsoln | soln | transmit | soln | back | fsoln |
| | ” | ” | bsoln | *reflect* | soln | back | bsoln |
| | ” | ” | bound | adsorb | soln | back | bound |
| | ” | bound | soln | desorb | bound | none | fsoln |
| action from | ” | ” | bsoln | desorb | bound | none | bsoln |
| bound state | ” | ” | bound | *no change* | bound | none | bound |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | ” | ” | bound’ | flip | bound | none | bound’ |
| | bound | soln | soln | *reflect* | bound | front | fsoln |
| | ” | ” | bsoln | transmit | bound | front | bsoln |
| | ” | ” | bound | hop | bound | front | bound |
| collision from | ” | ” | bound’ | hop | bound | front | bound’ |
| bound state | ” | bsoln | soln | transmit | bound | back | fsoln |
| | ” | ” | bsoln | *reflect* | bound | back | bsoln |
| | ” | ” | bound | hop | bound | back | bound |
| | ” | ” | bound’ | hop | bound | back | bound’ |
| | ” | bound | soln | desorb | bound | none | fsoln |
| action from | ” | ” | bsoln | desorb | bound | none | bsoln |
| bound state | ” | ” | bound | *no change* | bound | none | bound |
| | ” | ” | bound’ | flip | bound | none | bound’ |
| impossible | ” | bound’ | any | *nonsense* | none | none | none |

`void srfindex2tristate(enum MolecState ms3,enum PanelFace face,enum MolecState ms4,enum MolecState *msptr,enum MolecState *ms1ptr,enum MolecState *ms2ptr);`
Inverse function as `srftristate2index`. This function uses the same table as shown for `srftristate2index`, but in reverse. There are two ways that actions from bound states can be described using the tristate format, of which the former uses the default value of `ms` where it equals `MSsoln` and the latter uses the better value, which equals `ms1`. This function inverts to the better tristate values.

`int srfcompareaction(enum SrfAction act1,surfactionptr details1,enum SrfAction act2,surfactionptr details2)`
Compares the activity levels of `act1` and `act2`, returning -1 if `act1` is “more active”, 1 if `act2` is more active, and 0 if they are the same. If they have exactly the same activity levels, then this returns 0. If both `act1` and `act2` equal `SAmult`, then this turns to the details of these actions, entered in `details1` and `details2`, to determine which is more active. Here, the first difference that is encountered determines which is more active. For the actions, the “activity” of an action is ordered as: `SAtrans` ¡ `SAmult` ¡ `SAreflect` ¡ `SAjump` ¡ `SAabsorb` ¡ `SAport`.

**memory management**

`surfaceactionptr surfaceactionalloc(int species);`
Allocates a surface action structure for storing action details, and returns a pointer to it, or `NULL` if memory could not be allocated. Initializes most values to 0 or equivalent. Initializes `srfnewspec` to `species`.

`void surfaceactionfree(surfaceactionptr actdetails);`
Frees a surface action structure and its data.

`int panelsalloc(surfaceptr srf,int dim,int maxpanel,int maxspecies,enum PanelShape ps);`
Allocates `maxpanels` of shape `ps` for the surface `srf`; `srf` cannot be `NULL` but must be a surface. The `srf` element of the panels are set to `srf`. In `srf`, the correct `maxpanel` entry is set to `maxpanel`, the `npanel` entry is unchanged, and the proper list of panels are allocated and cleared. Also, panel names are created, each of which is set to the panel number, as a default. All points are set to all zeros. The function returns 1 for success and 0 for failure to allocate memory; if it fails, it does not do a good job of freeing working memory. The default jump destination is to the opposite side of the same panel.

This function may be called multiple times. It should be called when more panels of shape `ps` are needed. It *should not* be called if the only change is a larger value of `maxspecies`; in this case, a simple call to `surfacealloc` will take care of all required updates. This function calls `emittersalloc` to take care of the `emitterabsorb` panel element.

`void panelfree(panelptr pnl);`
Frees a single panel and all of its substructures (but not `srf`, because that's a reference and is not owned by the panel). This is called by `surfacefree` and so should not need to be called externally.

`int emittersalloc(surfaceptr srf,enum PanelFace face,int oldmaxspecies,int maxspecies);`
Allocates basic space for emitters (used for concentrations that match those for unbounded diffusion). This allocates the `srf->maxemitter` and `nemitter` arrays, as well as the first levels of the `emitteramount`, and `emitterpos` arrays. It also allocates all of the panel `emitterabsorb` arrays. Returns 0 for success or 1 for inability to allocate memory; in the latter case, this will create memory leaks.

There are three reasons to call this function: to set up the basics of emitters (send in `oldmaxspecies` as 0), to address emitter issues for new panels that had not been declared when emitters were set up originally (send in `oldmaxspecies` equal to `maxspecies`), or to allow for a larger `maxspecies` value (send in `oldmaxspecies` as whatever the prior `maxspecies` value was).

`int surfexpandmollist(surfaceptr srf,int newmax,int ll);` Allocates the surface molecule lists and expands them as needed. Returns 0 for success or 1 for failure to allocate memory. Enter `newmax` as the new maximum size for list number `ll`. Enter `ll` as -1 to set the number of molecule lists to `newmax` rather than to expand a single list.

`surfaceptr surfacealloc(surfaceptr srf,int oldmaxspecies,int maxspecies,int dim);`
Allocates a surface structure, and sets all elements to initial values. `maxspecies` is the maximum number of molecular species, which is used for allocating `action` and `actdetails`, as well as some emitter things. Colors are set to all 0's (black), but with alpha values of 1 (opaque); polygon modes are set to face if `dim` is 3, and to edge otherwise; `edgepoints` is set to 1; `action` is set to for transmitting (`SAtrans`) for the solution elements and to "no" (`SAno`) for the surface-bound elements. Panels and emitters are allocated here, although they are updated if necessary. This is called by `surfacessalloc` and so should not need to be called externally.

This function can be called multiple times. The two times when it might need to be called are when the surface does not exist and needs to be allocated, in which case send in `srf` as `NULL` and `oldmaxspecies` as 0, or when the surface does exist and `maxspecies` is being updated. In this case, send in the existing structure in `srf` and send in the prior number of maximum species in `oldmaxspecies`.

If this function is unable to allocate adequate memory, it returns `NULL`. If this is the case, it does not change any pre-existing `srf` data structure. In this case (which generally should not arise), it does not fully free the memory allocated here, leading to memory leaks.

`void surfacefree(surfaceptr srf,int maxspecies);`
Frees a surface, including all substructures and panels in it. This is called by `surfacessfree` and so should not need to be called externally. `maxspecies` is the number of molecule identities that the system was allocated with.

`surfacessptr surfacessalloc(surfacessptr srfss,int maxsurface,int maxspecies,int dim);`
Allocates a surface superstructure for `maxsurface` surfaces, as well as all of the surfaces. Each surface name is allocated to an empty string of `STRCHAR` (256) characters. Each surface is allocated for `maxspecies` species (a value of 0 is allowed). This function may be called more than once. On the first call, send in `srfss` as `NULL`, and the surface superstructure pointer will be returned; if it fails to allocate memory, it will return `NULL`. On subsequent calls, send in the existing surface superstructure pointer in `srfss` and the superstructure will be expanded as needed for the new larger `maxsurface` and/or `maxspecies` values (they may not be shrunk). In this case, the function will return the same pointer that was sent in, or `NULL` if it could not allocate memory; in the latter case, the original superstructure is unchanged. If this function is unable to allocate adequate memory (which generally should not arise), it does not fully free the memory allocated here, leading to memory leaks.

`void surfacessfree(surfacessptr srfss);`
Frees a surface superstructure pointed to by `srfss`, and all contents in it, including all of the surfaces and all of their panels.

**data structure output**

`void surfaceoutput(simptr sim);`
Prints out information about all surfaces, including the surface superstructure, each surface, and panels in the surface.

`void writesurfaces(simptr sim,FILE *fptr);`
Writes all information about all surfaces to the file `fptr` using a format that can be read by Smoldyn. This allows a simulation state to be saved.

`int checksurfaceparams(simptr sim,int *warnptr);`
Checks some surface parameters. Many more should be checked as well, although those haven't been written yet.

**structure set up**

`int surfenablesurfaces(simptr sim,int maxsurf);`
Allocates and sets up the surface superstructure for a maximum of `maxsurf` surfaces. This function may be called multiple times. If the input parameters are different at one call than from the previous call, then the surface superstructure is updated to the new parameters. Enter `maxsurf` as -1 to indicate default operation, meaning that nothing is done if the surface superstructure was allocated previously, or that 5 surfaces are allocated otherwise. Returns 0 for success (or nothing was done because it was done previously), 1 for failure to allocate memory, or 2 if `sim` is undefined.

`int surfexpandmaxspecies(simptr sim,int maxspecies);`
Expands the number of molecular species that the surfaces can work with, allocating memory as needed. Surfaces and molecules should be kept synchronized, so this should be called whenever the master maxspecies changes. Returns 0 for success or 1 for failure to allocate memory.

`surfaceptr surfaddsurface(simptr sim,char *surfname);`
Adds a surface called `surfname` to the simulation. This enables surfaces and/or allocates more surfaces, as necessary. A pointer to the surface is returned, or `NULL` is returned for failure to allocate memory. If `surfname` was already declared as a surface, then this function simply returns a pointer to the existing surface.

`void surfsetcondition(surfacessptr surfss,enum StructCond cond,int upgrade);`
Sets the surface superstructure condition to `cond`, if appropriate. Set `upgrade` to 1 if this is an upgrade, to 0 if this is a downgrade, or to 2 to set the condition independent of its current value. If the condition is downgraded, this also downgrades the simulation structure condition.

`int surfsetepsilon(simptr sim,double epsilon);`
Sets the `epsilon` value in the surface superstructure. Returns 0 for success, 2 if the surface superstructure did not exist and could not be created, or 3 for an illegal requested value ($\leq 0$).

`int surfsetmargin(simptr sim,double margin);`
Sets the `margin` value in the surface superstructure. Returns 0 for success, 2 if the surface superstructure did not exist and could not be created, or 3 for an illegal requested value ($\leq 0$).

`int surfsetneighdist(simptr sim,double neighdist);`
Sets the neighbor distance value in the surface superstructure. Returns 0 for success, 2 if the surface superstructure did not exist and could not be created, or 3 for an illegal requested value ($\leq 0$).

`int surfsetneighhop(surfaceptr srf,int neighhop);`
Sets the `neighhop` element of the surface structure to the value entered. Returns 0 unless `srf` is `NULL`, in which case it returns 1.

`int surfsetcolor(surfaceptr srf,enum PanelFace face,double *rgba);`
Sets the color vector for face `face` of surface `srf` to `rgba`. Any face value is allowed, including both and none. Returns 0 for success, 1 if no surface was entered, or 2 if one or more of the color or alpha values is out of range (between 0 and 1 inclusive).

`int surfsetedgepts(surfaceptr srf,double value);`
Sets the drawing thickness for surface `srf` to `value`. Returns 0 for success, 1 if no surface was entered, or 2 if `value` is negative.

`int surfsetstipple(surfaceptr srf,int factor,int pattern);`
Sets the stippling pattern for drawing surface `srf`. `factor`, which needs to be at least 1, is the repeat distance for the entire stippling pattern and `pattern`, which needs to be between 0 and 0xFFFF inclusive, is the pattern to be used. Enter either or both as negative values to not set that parameter. Returns 0 for success, 1 if the surface is undefined, or 2 if inputs are out of range.

`int surfsetdrawmode(surfaceptr srf,enum PanelFace face,enum DrawMode dm);`
Sets the surface drawing mode of face `face` of surface `srf` to `dm`. Any face value is allowed. The only drawing mode that is not allowed is `DMnone`. Returns 0 for success, 1 if the surface is undefined, or 2 if the drawing mode is out of range.

`int surfsetshiny(surfaceptr srf,enum PanelFace face,double shiny);`
Sets the shininess of face `face` of surface `srf` to `shiny`. Any face value is allowed. The shininess needs to be between 0 and 128, inclusive. Returns 0 for success, 1 if the surface is undefined, or 2 if the shininess is out of range.

`int surfsetaction(surfaceptr srf,int ident,const int *index,enum MolecState ms,enum PanelFace face,enum SrfAction act);`
Sets the `action` element of surface `srf`. `ident` is the molecule species, which should be a valid species index, -5 to indicate all species, or 0 to indicate a group of species, perhaps with a wildcard. If value 0 is used, then `index` should be sent in as the `index` element that gets returned by `molstring2index1`. `ms` is the species state, which should be a regular species state (not `MSbsoln`) or should be `MSall` to indicate all states. `face` is the interaction face of the surface, if any; it is allowed to adopt any value, including `PFfront`, `PFback`, `PFboth` (to indicate both front and back), or `PFnone` to indicate actions for surface-bound species that do not collide with other surfaces. `act` is the desired action. Returns 0 for success, 1 if `ident` is out of range, 2 if `ms` is out of range, or 3 if the requested action is not permitted for the indicated interaction face.

`int surfsetrate(surfaceptr srf,int ident,const int *index,enum MolecState ms,enum MolecState ms1,enum MolecState ms2,int newident,double value,int which);`
Sets the `srfrate` or `srfprob` element of the action details of surface `srf`, along with the `srfnewspec` and `srfdatasrc` elements. See the data table for the `srftristate2index` function for the possible input combinations of `ms`, `ms1`, and `ms2` and what they mean (this function uses the tristate input format). Enter `ident` as a positive number to indicate a specific species, -5 to indicate all species, or -6 to indicate wildcard selected species (channel 0). Typically, `newident` will be the same as `ident`, although it can be different for a species change at the surface. `value` is the desired rate or probability value. Enter `which` as 1 to set the `srfrate` element and as 2 to set the `srfprob` element. Returns 0 for success, 1 if `ident` is out of range (equal to 0), 2 if `ms` is out of range, 3 if `ms1` is out of range, 4 if `ms2` is out of range, 5 if `newident` is out of range, 6 if `value` is out of range, or -1 if memory could not be allocated for the surface action details data structure.

`int surfsetmaxpanel(surfaceptr srf,int dim,enum PanelShape ps,int maxpanel);`
Sets the maximum number of panels of shape `ps` for surface `srf` to `maxpanel`. The system dimensionality is `dim`. This function may be called multiple times. It allocates memory as needed.

`int surfaddpanel(surfaceptr srf,int dim,enum PanelShape ps,char *string,double *params,char *name);`
Adds or modifies a panel of shape `ps` to surface `srf`, in a `dim` dimensional system. `string` lists any text parameters for the panel, which in practice is only a single word that gives the orientation of a rectangle panel (e.g. "+0" or "-y"). `params` lists the numerical parameters for the panel location, size, and drawing characteristics. The number of necessary parameters can be found from the `surfpanelparams` function and the specific parameters, which depend on the panel shape and the system dimensionality, are described in the User Manual. `name` is an optional parameter; if it is included and is not an empty

string, the panel is named `name`. If this panel name was already used by a panel of the same shape, then this function overwrites that panel's data with the new data. If the name was already used by a panel with a different shape, then this creates an error, and if the name was not used before, then a new panel is created. To use default panel naming, send in `name` as either `NULL` or as an empty string.

This function returns 0 for success, -1 for inability to allocate memory, 1 for missing surface, 2 for `ps` out of range, 3 for unable to parse `string`, 4 for drawing slices or stack are zero or negative, 5 for cylinder ends are at same location, 6 for hemisphere outward pointing vector has zero length, 7 for a zero or negative radius, 8 for a normal vector with zero length, or 9 if the panel name was used before for a panel with a different shape.

This function calculates all of the geometrical data that are stored with the panel, such as the normal direction and the edge normals.

`int void surftransformpanel(panelptr pnl,int dim,double *translate,double *origin,double *expand);`

Performs translation and expansion of a surface panel, while maintaining the panel shape as much as possible. Enter the translation vector in `translate`, the origin coordinates about which the expansion should be performed in `origin`, and the expansion values for the different coordinates in `expand`. If expansion should be isotropic, enter the same values for all components of the `epxand` vector. Each component should be 1 for no expansion, less than 1 for retraction, and greater than 1 for expansion. Negative values indicate inversion. This function updates all panel `point` and `front` values. This lowers the surface condition, box condition, and compartment condition to indicate that some recomputation is needed. Note that Smoldyn cannot distort spheres, hemisphere, cylinders, or disks, so anisotropic expansion is only done approximately.

Most of this function is quite straightforward. Each panel point value that represents an actual coordinate is transformed through translation and expansion. Normals are recomputed as needed. Note that all computations are done in-place, which is usually easy, but some care is required to make sure that one is using the correct values. Several functions transform the radius of a panel. The math for this is as follows. Consider a 2D system, a radius vector (often given as `radiusv` in the code), $\mathbf{r}$, and suppose that the expansion vector is $(e_x, e_y)$. Also assume this radius vector starts at the expansion origin. In this case, the transformed vector endpoint will be at $(e_x r_x, e_y r_y)$. This means that the new radius will be $(e_x^2 r_x^2 + e_y^2 r_y^2)^{1/2}$. If the old radius vector has length $r$ and is parallel to the unit vector $\mathbf{n}$, then the new radius is $r(e_x^2 n_x^2 + e_y^2 n_y^2)^{1/2}$. For 3D objects, anisotropic expansion would normally distort shapes away from being perfect cylinders, hemispheres, spheres, or disks. However, Smoldyn does not support such distorted shapes, so this function just picks specific radius vectors and uses those.

`void surftranslatepanel(panelptr pnl,int dim,double *translate);`

Translates panel `pnl` by amount given in `translate`. No superstructure conditions are changed in this function. This function is similar to `surftransformpanel` but it much simpler.

`void surfupdateoldpos(surfaceptr srf,int dim);`

Copies the contents of each panel `point` element to the panel `oldpoint` element, and also each `front` element to the panel `oldfront` element. Use this when a surface is about to be moved and the `oldpoint` and `oldfront` elements should represent the prior location.

`void surftranslatesurf(surfaceptr srf,int dim,double *translate);`

Translates surface `srf` by amount given in `translate`. The box superstructure condition is downgraded to `SCparams`.

`int surfsetemitterabsorption(simptr sim);`

Sets emitter absorption probabilities for panels based on emitter information in the surface structures. Returns 0 for success or 1 if one or more of the distances between emitters and a surface panel was zero (which leads to divide-by-zero errors).

`int surfsetjumppanel(surfaceptr srf,panelptr pnl1,enum PanelFace face1,int bidirect,panelptr pnl2,enum PanelFace face2);`

Sets up jumping between face `face1` of panel `pnl1` and face `face2` of panel `pnl2`, for surface `srf`. Jumping is set up to be unidirectional, from `pnl1` to `pnl2` if `bidirect` equals 0 and is set up to go in both directions if `bidirect` equals 1. This only sets the panel jumping indices and does not assign jumping actions to any molecules. Returns 0 for success, 1 for no surface, 2 for no `pnl1`, 3 for `face1` out of range, 4 for `bidirect` out of range, 5 for an error with `pnl2` including it having a different shape from `pnl1` or it equaling `pnl1`, and 6 for `face2` out of range.

`double srfcalcrate(simptr sim,surfaceptr srf,int i,enum MolecState ms1,enum PanelFace face,enum MolecState ms2);`
Calculates the actual rate for the interaction of a molecule of type `i` and state `ms1` interacting with face `face` of surface `srf`, and ending up in state `ms2`. This uses the `srf->actdetails[i][ms][face]->srfprob[ms2]` data to calculate the actual conversion rate, and accounts for reversible interactions as much as possible. All cases are calculated assuming steady-state behavior, and all are found using the SurfaceParam.c function `surfacerate`. Returned rates will be between 0 and `MAX_DBL`. Error codes are returned with negative numbers: -1 indicates that the input situation is impossible (i.e. `ms1=MSsoln` and `face=PFnone`), or that input data are unavailable (i.e. the surface action isn't `SAmult` or the action details aren't recorded for this action); -2 indicates that the interaction probabilities haven't been computed yet in the action details structure, probably because the appropriate set up routine hasn't been called yet; and -3 indicates that reflection coefficients were requested, which cannot be computed here. See `srfcalcprob`.

`double srfcalcprob(simptr sim,surfaceptr srf,int i,enum MolecState ms1,enum PanelFace face,enum MolecState ms2);`
Calculates the surface interaction probability for the interaction of a molecule of type `i` and state `ms` interacting with face `face` of surface `srf`, and ending up in state `ms2`. This uses the `srf->actdetails[i][ms][face]->srfrate[ms2]` data to calculate the conversion probability, and accounts for reversible interactions as much as possible. All cases are calculated assuming steady-state behavior, and all are found using the SurfaceParam.c function `surfaceprob`. Returned probabilities will be between 0 and 1, inclusive, or an error code. Error codes are returned with negative numbers: -1 indicates that input data are unavailable (i.e. the surface action isn't `SAmult` or the action details aren't recorded for this action); -2 indicates that the rate is listed as being negative, which is impossible; and -3 indicates that reflection probabilities were requested, which cannot be computed here. See `srfcalcrate`.

`int surfsetneighbors(panelptr pnl,panelptr *neighlist,int nneigh,int add);`
Adds or removes neighbors to or from a panel's list of neighbors. `pnl` is the panel whose neighbor list should be modified, `neighlist` is a list of neighboring panels to be added or removed, `nneigh` is the number of neighbors that are listed in `neighlist`, and `add` is 1 if those listed in `neighlist` should be added, or 0 if they should be removed. For addition, neighbors are not added again if they are already in the list. If all neighbors should be removed, send `neighlist` in as `NULL`. This allocates space as needed. It returns 0 for success or 1 if not enough space could be allocated. For optimal memory allocation, it's slightly better if many neighbors are added at once in a single function call, rather than one neighbor per function call.

`int surfaddemitter(surfaceptr srf,enum PanelFace face,int i,double amount,double *pos,int dim);`
Adds an emitter to a surface so that it can be used for simulating unbounded diffusion. This takes care of any necessary memory allocating. `srf` is the surface that the emitter is being added to, `face` is the surface face, `i` is the species number, `amount` is the emitter amount, flux, or weighting, `pos` is the `dim`-dimensional position of the emitter, and `dim` is the system dimensionality. Returns 0 for success or 1 if this is unable to allocate memory. This does not calculate the panel absorption probabilities, but does allocate space for them, if needed.

`surfaceptr surfreadstring(simptr sim,surfaceptr srf,char *word,char *line2,char *erstr);`
Reads and processes one line of text from the configuration file, or some other source, for the surface pointed to by `srf`. If the surface is not known or has not been defined yet, then set `srf` to `NULL`.

The first word of the line should be sent in as `word` and the rest sent in as `line2`. If this function is successful, it returns the surface pointer and it does not change the contents of `erstr`; if not, it returns -1 and it writes an error message to `erstr`.

`int loadsurface(simptr sim,ParseFilePtr *pfpptr,char *line2,char *erstr);`
`loadsurface` loads a surface from an already opened disk file pointed to with `fptr`. `lctrptr` is a pointer to the line counter, which is updated each time a line is read. If successful, it returns 0 and the surface is added to the surface superstructure in `sim`, which should have been already allocated. Otherwise it returns the updated line counter along with an error message. If a surface with the same name (entered by the user) already exists, this function can add more panels to it. It can also allocate and set up a new surface. If this runs successfully, the complete surface structure is set up, with the exception of box issues. If the routine fails, any new surface structure is freed.

`int surfupdateparams(simptr sim);`
Sets the simulation time step for surface parameters. This includes setting the neighbor distance (`srf->neighdist`) to 0.1 times the longest surface-bound diffusion rms step-length, and setting surface interaction probabilities (`srf->prob`). All probabilities are either simply set to 0 or 1 or are set to an intermediate value with the SurfaceParam.c function `srfprob`. The latter ones account for reversible or competing processes, as appropriate. They are cumulative probabilities. Returns 0 for success or 2 if the molecules aren't adequately set up.

`int surfupdatelists(simptr sim);`
Sets up surface molecule lists, area lookup tables, and action probabilities. If calculated probabilities exceed 1 or add up to more than 1, they are adjusted as needed, although this may affect simulation results. No warnings are returned about these possible problems, so they should be checked elsewhere. Returns 0 for success, 1 for inability to allocate memory, or 2 for molecules not being sufficiently set up beforehand. This function may be called at setup, or later on during the simulation.

`int surfupdate(simptr sim);`
Sets up or updates surface data structures.

**core simulation functions**

`enum PanelFace panelside(double* pt,panelptr pnl,int dim,double *distptr,int strict,int useoldpos);`
Returns the side of the panel `pnl` that point `pt` is on. If `strict` is 0, then `PFback` is returned if the point is exactly at the panel, while if `strict` is 1 then `PFnone` is returned if the point is exactly at the panel. In general, this should not be set for strict use when getting panel faces and should be for strict use for assistance in setting panel faces. If `distptr` is sent in as a non-`NULL` pointer, its contents will be set to the distance that `pt` is away from the infinite panel, with a positive number for the front side and negative or zero for the back. The values returned by this function define the side that `pt` is on, so should be called for other functions that care. The distance value that is returned by this function is also used to determine the face; thus, if two calls with different `pt` values return the exact same distance value, then the same face will be returned. Set `useoldpos` to 1 if the old point and front values should be used for the panel position rather than the current point and front values.

`void panelnormal(panelptr pnl,double *pos,enum PanelFace face,int dim,double *norm);`
Returns, in `norm`, the normal vector for the panel `pnl`, that points outwards from the `face` side. If this is a curved panel, such as a sphere or a cylinder, then `pos` is the position on the surface for which the local normal should be computed. If `face` is not equal to `PFfront` or `PFback`, then it is assumed that the front side is desired.

`int lineXpanel(double *pt1,double *pt2,panelptr pnl,int dim,double *crsspt,enum PanelFace *face1ptr,enum PanelFace *face2ptr,double *crosspt r,double *cross2ptr,int *veryclose,int useoldpoint);`
This determines if the line from `pt1` to `pt2` crosses the panel `pnl`, using a `dim` dimensional system. These input variables are not changed by this function; other variables are for output only such that

prior values are not looked at by the function. With those, `crsspt` must be a `dim` dimensional vector and the others may be `NULL` or can be pointers to single values that will be overwritten. The panel includes all of its edges. 1 is returned if the line crosses the panel at least once and 0 if it does not. If it does not cross, all return values except `veryclose` are undefined. If it crosses, `crosspt` will be the coordinates of the crossing, `face1` will be the side of the panel that is first impacted, `face2` will be the side that is towards `pt2`, and the contents of `crossptr` will be the crossing position on the line, which is between 0 and 1 inclusive. The type of crossing can be determined by looking at the returned `face` values. (1) If `face1!=face2`, then the line crosses the panel exactly once and the contents of `cross2` are undefined. (2) if `face1==face2`, then the line crosses the panel exactly twice (implying a curved panel) for which the first crossing will be recorded in `crsspt` and `crossptr` and the latter in `cross2ptr`. Suppose the line crosses a hemisphere in such a way that it enters the corresponding sphere where the hemisphere is open and departs where it is a panel. In this case, the `face1` value will equal the inside face (unlike the value returned by `panelside`). In contrast, if it enters on the closed side and exits on the open side, both `face1` and `face2` will equal the outside. The same rules apply for cylinders.

`veryclose`, if it is entered as non-`NULL`, is returned equal to 1 if `pt1` is within `VERYCLOSE` distance units of the panel, as 2 if `pt2` is this close to the panel, as 3 if both points are this close to the panel, and as 0 if neither point is this close to the panel. These codes can be used to determine if round-off error is likely to be a problem.

Set `useoldpos` if the `oldpoint` and `oldfront` elements should be used for the panel position rather than the current location given in `point` and `front`.

While `crsspt` will be returned with coordinates that are very close to the panel location, it may not be precisely at the panel, and there is no certainty about which side of the panel it will be on; if it matters, fix it with `fixpt2panel`. Similar rules apply for the contents of `crossptr` and `cross2ptr`.

Each portion of this routine does the same things, and usually in the same order. Crossing of the infinite panel is checked, the crossing value is calculated, the crossing point is found, and finally it is determined if intersection actually happened for the finite panel. For hemispheres and cylinders, if intersection does not happen for the first of two possible crossing points, it is then checked for the second point.

`int lineexitpanel(double *pt1,double *pt2,panelptr pnl,int dim,double *pnledgept,int *exitside)`

This finds where the line segment that goes from `pt1` to `pt2` exits the panel `pnl`. Although it probably isn't essential, this function assumes that the line segment is nearly co-planar with its local position on the panel (or parallel to the panel for 2D). It returns the answer as coordinates in `pnledgept`. This works in 1, 2, or 3-D. This returns the side or axis of the panel that the line exits through in `exitside`. See below. Returns 0 for success or 1 if `pt1` is identical to `pt2`, which is an error.

For the most part, this function calls other functions for each of the possibilities. An exception is for 2D cylinders, which are really just 2 parallel lines. In this case, this function computes the dot product between the vector from a cylinder end to `pt1` and the cylinder normal vector, using the sign of the dot product to determine which line to consider. Then, it calls `Geo_LineExitLine2` using this line. This function was new in Smoldyn 2.37.

| 1D | 2D | 3D |
|---|---|---|
| **rectangles, ps = PSrect** | | |
| 1 | 1 for `p[0]` | 1 for `p[0]` - `p[1]` |
| | 2 for `p[1]` | 2 for `p[1]` - `p[2]` |
| | | 3 for `p[2]` - `p[3]` |
| | | 4 for `p[3]` - `p[0]` |
| **triangles, ps = PStri** | | |
| 1 | 1 for `p[0]` | 1 for `p[0]` - `p[1]` |
| | 2 for `p[1]` | 2 for `p[1]` - `p[2]` |
| | | 3 for `p[2]` - `p[3]` |

| | | |
|---|---|---|
| **spheres, ps = PSsph** | | |
| 1 | 1 | 1 |
| **cylinders, ps = PScyl** | | |
| not allowed | 1 for `p[0]`<br>2 for `p[1]` | 1 for `p[0]`<br>2 for `p[1]` |
| **hemispheres, ps = PShemi** | | |
| not allowed | 1 for cw end<br>2 for ccw end | 1 |
| **disks, ps = PSdisk** | | |
| not allowed | 1 for end cw of normal<br>2 for end ccw of normal | 1 |

`void paneledgenormal(panelptr pnl,double *pnledgept,int dim,int edgenum,double *normal)`
Returns the outward-pointing unit normal vector to the edge of panel `pnl`, where this normal is plane-parallel to the local panel surface and perpendicular to the panel edge. In 2D, this normal is parallel to the panel at its edge. This normal points out of the panel. Send in `pnledgept` as a point that is at the edge of the panel (not always used) and `edgenum` as the number of the edge (not always used). See the table for `lineexitpanel` for the correct values to use for this input. The result is returned in the vector `normal`. If `edgenum` is entered as zero, then this implies that the point is not at a panel edge but in the center of the panel somewhere. In this case, this returns a vector that is parallel to the surface at `pnledgept`. It is not specified here in which direction this vector points, except simply that it is parallel to the surface at the point `pnledgept`.

Most portions of this function are fairly straightforward. This function only works for 2D and 3D, because it doesn't make sense in 1D. This function was new in version 2.37.

The sphere portion is written under the assumption that `edgenum` is equal to zero, because there are no edges to this surface.

`int ptinpanel(double *pt,panelptr pnl,int dim);`
Determines if the point `pt` is inside the finite panel `pnl`, returning 1 if so and 0 if not. Here, inside only means that the point is within the volume that is swept out perpendicular to the plane of the panel, and says nothing about the position of the point relative to the plane of the panel.

This function is nearly identical to the portion of `lineXpanel` that checks whether the position `crsspt` is within the panel or not.

`enum SrfAction surfaction(surfaceptr srf,enum PanelFace face,int ident,enum MolecState ms,int *i2ptr,enum MolecState *ms2ptr);`
Returns the surface action that should happen to a molecule of type `ident` and state `ms` that interacts with face `face` of surface `srf`. `ms` needs to be a real molecule state, meaning that it is not allowed to be `MSbsoln`. If the state of `ident` should be changed, then the new state is returned in `ms2ptr`, if `ms2ptr` is not `NULL` (`ms2ptr` may be returned pointing to `MSbsoln`, as well as to `MSsoln`). If the species of `ident` should be changed, then the new species number is returned in `i2ptr`, if `i2ptr` is not `NULL`; if it should not be changed, then `i2ptr` is returned pointing to the same value as `ident`. This function does not return `SAmult`; instead, it specifies what should happen in detail, including `SAadsorb` for adsorption, `SArevdes` for reversible desorption, `SAirrevdes` for irreversible desorption, and `SAflip` for on-surface state change.

`int rxnXsurface(simptr sim,moleculeptr mptr1,moleculeptr mptr2);`
Returns 1 if a potential bimolecular reaction between `mptr1` and `mptr2` is across a non-transparent surface, and so cannot actually happen. Returns 0 if a reaction is allowed. Using the diffusion coefficients of the two molecules, this calculates the reaction location and then determines which molecules need to diffuse across which surfaces to get to that location. If the molecules can diffuse across the necessary surfaces, then the reaction is allowed, and not otherwise. This routine does not allow reactions to occur across jump surfaces. Also, it does not look for jump paths that go from `mptr1`

to `mptr2`. Surface-bound molecules that are in their "up" or "down" state are assumed to be accessible from both sides of the surface, whereas those that are in the "front" or "back" states are accessible from only one side.

`void getpanelnormal(double *pt,panelptr pnl,int dim,double *norm);`
Finds the local panel normal for panel `pnl` at position `pt`, returning the vector in `norm`. The result points towards the front side of the panel. This only considers the infinite plane of the panel, while ignoring its boundaries (similarly, hemispheres are considered to be identical to spheres and cylinders are considered to be infinitely long).

`void fixpt2panel(double *pt,panelptr pnl,int dim,enum PanelFace face,double epsilon);`
Fixes the point `pt` onto the face `face` of panel `pnl`. Send in `face` equal to `PFnone` if `pt` should be moved as close as possible to `pnl`. If it should also be on the front or back face of the panel, as determined by `panelside`, then send in `face` equal to `PFfront` or `PFback`, respectively. Before moving, if `pt` is already on the proper face and its distance is less than or equal to `epsilon`, it is not moved; setting `epsilon` to 0 ensures moving. This function first moves `pt` to the panel in a direction normal to the local panel surface and then nudges `pt` as required to get it to the proper side. This only considers the infinite plane of the panel, while ignoring its boundaries (similarly, hemispheres are considered to be identical to spheres and cylinders are considered to be infinitely long).

`int fixpt2panelnocross(simptr sim,double *pt,panelptr pnl,int dim,enum PanelFace face,double epsilon);`
This is very similar to `fixpt2panel`, in that it moves point `pt` to the correct side of panel `pnl` based on the value of `face`. The difference is that this also ensures that no other panels are crossed in the process. If other panels are crossed, then this keeps on adjusting the point until no other panels are crossed. The function usually returns 0 but will return 1 if it tried 20 times to find a suitable fixed position and was unable to do so; in this latter case, the point is not moved at all. Any crossings of the panel `pnl`, and all of its neighboring panels are ignored.

`void movept2panel(double *pt,panelptr pnl,int dim,double margin);`
This moves the point `pt` to the nearest location that it is over the panel `pnl`, and also inside the edge by distance `margin`. This means that `pt` is not moved into the plane of the panel, which is done by `fixpt2panel`, but is moved parallel to the plane of the panel.

`int closestpanelpt(panelptr pnl,int dim,double *testpt,double *pnlpt,double margin);`
Finds the closest point that is on panel `pnl` to the test point `testpt` and returns it in `pnlpt`. `testpt` and `pnlpt` should not point to the same memory address. This returns an integer which is 0 if the closest panel point is not on the panel edge, and which is the edge number if the closest panel point is on an edge. The edge numbers are listed in the table following the `lineexitpanel` description. If the test point is exactly at the panel edge, then the edge number is returned.

If the closest panel point is within `margin` of the edge or a corner, even if the test point is technically within the panel, then the coordinates at the edge or corner are returned.

`double closestsurfacept(surfaceptr srf,int dim,double *testpt,double *pnlpt,panelptr *pnlptr,boxptr bptr);`
Finds the closest point that is on the surface `srf` to the test point `testpt` and returns it in `pnlpt` if `pnlpt` is not `NULL`.. This also returns the panel that that point is on, in `pnlptr`, if `panelptr` is not `NULL`. It returns the distance between `testpt` and `pnlpt`, which should always be positive. If this returns a negative number, that means that the surface has no panels. If `bptr` is entered as `NULL`, then this searches over all panels on the listed surface, but if `bptr` is entered pointing to a box, then this only searches the panels that are in that box. `pnlpt` is allowed to be the same pointer as `testpt`.

`void movemol2closepanel(simptr sim,moleculeptr mptr,int dim,double epsilon,double neighdist,double margin);`
Checks to see if surface-bound molecule `mptr` is within the area of the finite panel `mptr->pnl` (i.e. over or under the panel, ignoring the position relative to the plane of the panel). If it isn't, this sees if

mptr is over a neighboring panel and if so, this puts mptr->pos on the neighboring panel at the correct location. At the end, and regardless of whether the molecule changed panels or not, the molecule position is fixed to the correct face of its panel using fixpt2panel. If the molecule needs to be moved parallel to the plane of a panel, whether back to its original panel or onto a new panel, then it is inset from the edge by distance margin. This function should only be called if the system dimensionality is 2 or 3.

This function is called by the diffuse function for putting surface-bound molecules in the correct places after they are diffused. It is also called by the dosurfinteract function after a surface-bound molecule reflects off of another surface. It was re-written in July-September of 2015, for Smoldyn version 2.37, because Boris Slepchenko discovered that its prior design produced inaccurate diffusion. In the new version, the molecule is projected onto its infinite panel according to the local panel normal. If it is then within the finite panel, the function is done. Otherwise, the function determines the coordinates where the molecule's trajectory exits the panel, it picks a random neighboring panel that is close to these coordinates to be the new panel, and the molecule's trajectory is rotated over to the new panel. If there is no new panel, then the molecule bounces off of the edge of the current panel. The function repeats this process until the full molecule trajectory is used up.

```
void surfacereflect(moleculeptr mptr,panelptr pnl,double *crsspt,int dim,enum PanelFace
    face);
```

This bounces the molecule mptr off of the face face side of panel pnl. Elastic collisions are performed, which should work properly for any shape panel and any dimensionality. For flat panels, elastic collisions also apply to Brownian motion. It is assumed that the molecule travels from some point (not given to this function, and irrelevant) to mptr->pos, via a collision with the panel at location crsspt, where crsspt is either exactly at the panel or is slightly on impact side of the panel. The molecule pos element is set to the new, reflected, position, which will always be on the face side of the panel.

```
int surfacejump(moleculeptr mptr,panelptr pnl,double *crsspt,enum PanelFace face,int
    dim);
```

This performs a jump for molecule mptr that hit panel pnl on face face. The contact location is input in crsspt, which needs to be very close to the panel but does not have to be on the proper side. This looks up the jump destination and translates both the crsspt value and the molecule position in mptr->pos to represent this jump. On return, crsspt is on the destination face of the destination panel. The molecule pos element will always be returned on the destination face side of the destination panel. For the most part, this function only allows jumps between panels with the same shape, the same dimensions, and the same orientation. Exceptions are that sphere, hemisphere, and cylinder radii are allowed to differ between origin and destination panels. This function works for any molecule state. Returns 0 if no jump happened (i.e. pnl->jumpp[face] was NULL or pnl->jumpf[face] wasn't either PFfront or PFback) and returns 1 if a jump happened.

Internally, for each surface, a few things are calculated. delta is the jump offset (jumped position minus current position) and dir is the relative orientation of the panels (1 if parallel, -1 if antiparallel). The jump offset is then added to crsspt and to mptr->pos, and subtracted from mptr->posoffset; the final thing means that pos+posoffset is always the diffused to position, and does not include any jumps. At the end, the crsspt and molecule position are finalized.

```
int dosurfinteract(simptr sim,moleculeptr mptr,int ll,int m,panelptr pnl,enum PanelFace
    face,double *crsspt);
```

Performs interaction between molecule and surface for a collision that is known to have happened or for possible interaction from a surface-bound state. This converts, kills, reflects, adsorbs, desorbs, etc. molecules, as appropriate. This function is called in three situations: (1) by checksurfaces when a molecule is found to have diffused across a surface, in which case pnl is the panel that was diffused across and face is the fact that was diffused into, (2) by checksurfacebound when a molecule is surface-bound, and thus might be able to desorb or flip, in which case pnl is the panel to which the molecule is bound and face is PFnone, and (3) by checksurfaces1mol which is used by the lattice hybrid software to check the surfaces for a single molecule; in this case, ll and m are set to -1. Quite possibly, the code would be better if these uses were separated into independent functions, but that's

not the case at the moment. This function is sometimes called twice for the same molecule during a single time step, if that molecule is both surface-bound and it diffuses across a different panel, but that should not lead to incorrect rates due to the different input parameters for the two calls.

At the beginning of the function, a few lines take care of special cases. The `isneigh` test, which only applies to situation (1) above, determines if the molecule is surface-bound and it diffused across a neighboring panel. If this test returns true, then the molecule has equal odds of staying on the same panel or crossing to the new panel. This is used so that molecules can diffuse from, say, one sphere to a neighboring and intersecting sphere, despite the fact that neither sphere has an open edge. This special case doesn't really address a proper surface interaction, but instead enables diffusion from panel to panel on a single surface. The next test is also for situation (1) and is for collision with unbounded emitter type surfaces. If neither special case holds, the action is gotten from `surfaction`. This function again copes with the two separate situations by testing whether `face` is `PFnone`.

On return, `crsspt` will be on the same side of the surface as the molecule. Returns 1 if the molecule does not need additional trajectory tracking (e.g. it's absorbed) and 0 if it might need additional tracking (e.g. it's reflected). This function does not consider opposite-face actions. For example, if the front of a surface is transparent and the back is absorbing, an impact on the front will result in the molecule being transmitted to the far side, and not being absorbed.

`int checksurfaces1mol(simptr sim,moleculeptr mptr,double crossminimum);`

Essentially identical to `checksurfaces` function, but just for a single molecule. Also, and very importantly, this assumes that a molecule's trajectory starts at `mptr->via` and not at `mptr->posx`. The reason is that this function was designed for checking surfaces after a molecule hit a port, so it's for the surfaces that are after the "via" position. If the molecule's list specifies that the molecule should be in a port buffer, even if it isn't there already, then the live list is not updated further to reflect new surface interactions. However, the molecule will still change states, get killed, etc. as appropriate. The `crossminimum` value is used to indicate that a crossing should be ignored unless its value is greater than the `crossminimum` value. As usual, the value is the distance along the molecule's straight-line trajectory where the crossing occurs, where 0 is the starting point and 1 is the ending point. This check adds a distance of `VERYCLOSE` to the `crossminimum` value to prevent unintentional cross determinations due to round-off error.

Returns 0.

`int checksurfaces(simptr sim,int ll,int reborn);`

Takes care of interactions between molecules and surfaces that arise from diffusion. Molecules in live list `ll` are considered; if `reborn` is 1, only the reborn molecules of list `ll` are considered. This transmits, reflects, or absorbs molecules, as needed, based on the panel positions and information in the molecule `posx` and `pos` elements. Absorbed molecules are killed but left in the live list with an identity of zero, for later sorting. Reflected molecules are bounced and their `posx` values represent the location of their last bouncing point. This function does not rely on molecules being properly assigned to boxes, and nor does it assign molecules to boxes afterwards. However, it does rely on the panels being properly assigned to boxes. If multiple surfaces are coincident, only the last one is effective. Returns error code of 0.

This function includes two "hacks." First, if a molecule has over 50 surface interactions during the same diffusion step, this function decides that something has gone wrong, and it simply puts the molecule back to where it started and moves on to deal with the next molecule. I'm not aware that this option has ever happened, but it's here because I suspect that it's possible for a molecule to become trapped in an endless loop.

The other "hack" is that this function looks for both the first and second surface crossings along the molecule's current trajectory. If they differ by less than $10^{-12}$ (but the difference is greater than zero), this function doesn't bother dealing with either surface, but puts the molecule back to its last known good position. The idea is that if the difference equals 0, then the last declared surface panel has priority. However, if the difference is negligibly larger than zero, then round-off errors are likely to dominate for calculations, which can cause the molecule to accidentally cross one of the two surfaces.

The only time that this hack is likely to become a problem is if the user defines two essentially coincident surfaces, in which case they will become effectively reflective.

`int checksurfacebound(simptr sim,int ll);`
Takes care of actions for surface-bound molecules, such as desorption, on-surface orientation flipping, etc. Returns error code of 0.

## 4.7 Boxes (functions in smolboxes.c)

The simulation volume is exactly divided into an array of identical virtual boxes. These allow the simulation to run efficiently because only potential reactions between molecules that are known to be physically close need to be checked, and the same for molecule-surface interactions. In principle, the boxes are fairly simple. In practice though, they complicate the overall code quite significantly. While the boxes are sized to exactly fill the simulation volume, the edge boxes are considered to extend beyond the volume to plus or minus infinity. In this way, all of space is within some box, and points outside of the simulation volume are assigned to the nearest box. Each box has its own box structure.

```
typedef struct boxstruct {
  int *indx;                    // dim dimensional index of the box [d]
  int nneigh;                   // number of neighbors in list
  int midneigh;                 // logical middle of neighbor list
  struct boxstruct **neigh;     // all box neighbors, using sim. accuracy
  int *wpneigh;                 // wrapping code of neighbors in list
  int nwall;                    // number of walls in box
  wallptr *wlist;               // list of walls that cross the box
  int maxpanel;                 // allocated number of panels in box
  int npanel;                   // number of surface panels in box
  panelptr *panel;              // list of panels in box
  int *maxmol;                  // allocated size of live lists [ll]
  int *nmol;                    // number of molecules in live lists [ll]
  moleculeptr **mol;            // lists of live molecules in the box [ll][m]
  } *boxptr;
```

`boxstruct` (declared in smollib.h) is a structure for each of the virtual boxes that partition space. Each box has a list of its neighbors, in `neigh`, as well as a little information about them. This list extends from 0 to `nneigh-1`. From 0 to `midneigh-1` are those neighbors that logically precede the box, meaning that they are above or to the left, whereas those from `midneigh` to `nneigh-1` logically follow the box. If there are no periodic boundary conditions, the logical order is the same as the address order; however, this is not necessarily true with the inclusion of wrap-around effects. In `wpneigh` is a code for each neighbor that describes in what way it is a neighbor: 0 means that it's a normal neighbor with no edge wrap-around; otherwise pairs of bits are associated with each dimension (low order bits for low dimension), with the bits equal to 00 for no wrapping in that dimension, 01 for wrapping towards the low side, and 10 for wrapping towards the high side. This might be clearer in the Zn.c documentation. The neighbors that are listed depend on the requested simulation accuracy:

| accuracy | neighbors | wrap-around |
|---|---|---|
| <3 | none | no |
| 3 to <6 | nearest | no |
| 6 to <9 | nearest | yes |
| <9 | all | yes |

Boxes also have lists of molecules, allocated to size `maxmol[ll]` and filled from 0 to `nmol[ll]-1`) that correspond to the master molecule lists, and walls (`wlist`, allocated and filled with `nwall` pointers) within them. While the lists are owned by the box, the members of the lists are simply references, rather than implications of ownership. The same, of course, is true of the neighbor list, although the box owns the `wpneigh` list. If wall or neighbor lists are empty, the list is left as `NULL`, whereas the molecule list always has

a few spaces in it. Boxes are collected in a box superstructure.

```
typedef struct boxsuperstruct {
  enum StructCond condition;    // structure condition
  struct simstruct *sim;        // simulation structure
  int nlist;                   // copy of number of molecule lists
  double mpbox;                // requested number of molecules per box
  double boxsize;              // requested box width
  double boxvol;               // actual box volumes
  int nbox;                   // total number of boxes
  int *side;                   // number of boxes on each side of space
  double *min;                 // position vector for low corner of space
  double *size;                // length of each side of a box
  boxptr *blist;               // actual array of boxes
  } *boxssptr;
```

`boxsuperstruct` (declared in smollib.h) expresses the arrangement of virtual boxes in space, and owns the list of those boxes and the boxes. `condition` is the current condition of the superstructure and `sim` is a pointer to the simulation structure that owns this superstructure. `nlist` is a copy of the number of molecule lists that are used in the molecule superstructure. This is used here, and the `mol` element of the individual boxes are allocated to this, rather than `maxlist`, because boxes can potentially use up lots of memory, and this saves allocating memory unnecessarily.

Either `mpbox` or `boxsize` are used but not both. Boxes are arranged in a rectangular prism grid and exactly cover all space inside the walls. The structure of the boxes in space is the same as that of a `dim` rank tensor, allowing tensor indexing routines to be used to convert between box addresses and indices. The box index along the `d`'th dimension of a point with position `x[d]` is

```
indx[d]=(int)((x[d]-min[d])/size[d]);
```

where integer conversion takes care of the truncation. Because of this, a box includes the points that are exactly on the low edge, but not those that are exactly on the high edge. Converting from box index to address is easy with the tensor routine in Zn.c, or can also be calculated quickly with the following code fragment, which outputs the box number as b,

```
for(b=0,d=0;d<dim;d++)    b=side[d]*b+indx[d];
```

Converting the box number to the indices can also be done, but the Zn.c routine is easiest for this.

**low level utilities**

`void box2pos(simptr sim,boxptr bptr,double *poslo,double *poshi);`
Given a pointer to a box in `bptr`, this returns the coordinate of the low and/or high corners of the box in `poslo` and `poshi`, respectively. They need to be pre-allocated to the system dimensionality. If either point is unwanted, enter `NULL`. This requires that the `min` and `size` portions of the box superstructure have been already set up.

`boxptr pos2box(simptr sim,double *pos);`
`pos2box` returns a pointer to the box that includes the position given in `pos`, which is a `dim` size vector. If the position is outside the simulation volume, a pointer to the nearest box is returned. This routine assumes that the entire box superstructure is set up.

`void boxrandpos(simptr sim,double *pos,boxptr bptr);`
Returns a uniformly distributed random point, in `pos`, that is within the box `bptr`.

`int panelinbox(simptr sim,panelptr pnl,boxptr bptr);`
Determines if any or all of the panel `pnl` is in the box `bptr` and returns 1 if so and 0 if not. For most panel shapes, this is sufficiently complicated that this function just calls other functions in the library file Geometry.c.

`int boxaddmol(moleculeptr mptr,int ll);`
Adds molecule `mptr`, which belongs in live list `ll`, to the box that is pointed to by `mptr->box`. Returns 0 for success and 1 if memory could not be allocated during box expansion.

`void boxremovemol(moleculeptr mptr,int ll);`
Removes molecule `mptr` from the live list `ll` of the box that is pointed to by `mptr->box`. Before returning, `mptr->box` is set to `NULL`. If the molecule is not in the box that's listed, then this doesn't try removing it; this result is fine if the molecule is actually in no box at all (which can happen) but is probably a bug if the molecule is in the wrong box.

`boxptr boxscansphere(simptr sim,const double *pos,double radius,boxptr bptr,int *wrap);`
Allows the calling function to scan over all of the boxes that are at least partially within `radius` distance of the point `pos`. This accounts for system edges and periodic boundaries. Call this at the first time in a scan with `bptr` equal to `NULL`, which instructs the function to set up internal static variables, and in subsequent scans with `bptr` not equal to `NULL`. This returns boxes repeatedly for the scan, and then returns `NULL` when the scan is complete. At present, it is not possible to skip ahead in the scan using the `bptr` input, although this could be implemented if it would be useful.

If the system has periodic boundaries, this returns boxes that are found upon wrapping, possibly returning the same box multiple times if it is within `radius` of `pos` in multiple ways. The amount of wrapping on each axis is returned in `wrap`, which needs to be entered as a non-`NULL` vector.

**memory management**

`boxptr boxalloc(int dim,int nlist);`
`boxalloc` allocates and minimally initiallizes a new `boxstruct`. Lists allocated are `indx`, which is size `dim`, and `maxmol`, `nmol`, and `mol`, each of which are size `nlist`. `nlist` may be entered as 0 to avoid allocating the latter lists. No molecule spaces are allocated.

`int expandbox(boxptr bptr,int n,int ll);`
Expands molecule list `ll` within box `bptr` by `n` spaces. If `n` is negative, the box is shrunk and any molecule pointers that no longer fit are simply left out. This function may be used if the initial list size (`bptr->maxmol[ll]`) was zero and can also be used to set the list size to zero. The book keeping elements of the box are updated. The function returns 0 if it was successful and 1 if there was not enough memory for the request.

`int expandboxpanels(boxptr bptr,int n);`
Expands the list of panels in box `bptr` by `n` spaces. If $n \leq 0$, this function ignores it, and does not shrink the box. This updates the `maxpanel` element. Returns 0 for success and 1 for insufficient memory.

`void boxfree(boxptr bptr,int nlist);`
Frees the box and all of its lists, although not the structures pointed to by the lists. `nlist` is the number of live lists.

`boxptr *boxesalloc(int nbox,int dim,int nlist);`
`boxesalloc` allocates and initializes an array of `nbox` boxes, including the boxes. `dim` is the system dimensionality and `nlist` is the number of live lists. There is no additional initialization beyond what is done in `boxalloc`.

`void boxesfree(boxptr *blist,int nbox,int nlist);`
Frees an array of boxes, including the boxes and the array. `nlist` is the number of live lists.

`boxssptr boxssalloc(int dim);`
Allocates and initializes a superstructure of boxes, including arrays for the `side`, `min`, and `size` members, although the boxes are not added to the structure, meaning that `blist` is set to `NULL` and `nbox` to 0.

`void boxssfree(boxssptr boxs);`
Frees a box superstructure, including the boxes.

**data structure output**

`void boxoutput(boxssptr boxs,int blo,int bhi,int dim);`
This displays the details of virtual boxes in the box superstructure `boxs` that are numbered from `blo` to `bhi-1`. To continue to the end of the list, set `bhi` to -1. This requires the system dimensionality in `dim`.

`void boxssoutput(simptr sim);`
Displays statistics about the box superstructure, including total number of boxes, number on each side, dimensions, and the minimum position. It also prints out the requested and actual numbers of molecules per box.

`int checkboxparams(simptr sim,int *warnptr);`
Checks and displays warning about various box parameters such as molecules per box, box sizes, and number of panels in each box.

**structure set up**

`void boxsetcondition(boxssptr boxs,enum StructCond cond,int upgrade);`
Sets the box superstructure condition to `cond`, if appropriate. Set `upgrade` to 1 if this is an upgrade, to 0 if this is a downgrade, or to 2 to set the condition independent of its current value. If the condition is downgraded, this also downgrades the simulation structure condition.

`int boxsetsize(simptr sim,const char *info,double val);`
Sets the requested box size. `info` is a string that is "molperbox" for the `mpbox` element, or is "boxsize" for the `boxsize` element, and `val` is the requested value. If the box superstructure has not been allocated yet, this allocates it. Returns 0 for success, 1 for failure to allocate memory, 2 for an illegal value, or 3 for the system dimensionality has not been set up yet.

`int boxesupdateparams(simptr sim);`
Creates molecule lists for each box and sets both the box and molecule references to point to each other.

`int boxesupdatelists(simptr sim);`
Sets up a superstructure of boxes, and puts some things in them boxes, including wall references. It sets up the box superstructure, then adds indices to each box, then adds the box neighbor list along with neighbor parameters, then adds wall references to each box. The function returns 0 for successful operation, 1 if it was unable to allocate sufficient memory, 2 if required things weren't set up yet. This function can be very computationally intensive.

`int boxesupdate(simptr sim);`
Sets up or updates box data structures.

**core simulation functions**

`boxptr line2nextbox(simptr sim,double *pt1,double *pt2,boxptr bptr);`
Given a line segment which is defined by the starting point `pt1` and the ending point `pt2`, and which is known to intersect the virtual box pointed to by `bptr`, this returns a pointer to the next box along the line segment. If the current box is also the final one, `NULL` is returned. Virtual boxes on the edge of the system extend to infinity beyond the system walls, so this function accurately tracks lines that are outside of the system volume.

This function would be simple, except that it has to cope with a few fairly rare exceptions. In particular, if one of the points is exactly in line with a corner between boxes (`flag` equals 1 or 2), it has to deal with it correctly, which is what the `if(flag)` portion copes with. As mentioned above, boxes include their low sides, but not their high sides. Thus if the destination is towards a corner point that is not in the current box, then the line should first visit a direction that is increasing, followed by a direction that is decreasing; if multiple directions increase, the trajectory moves diagonally, and if all directions are decreasing (`flag` equals 2) then the trajectory also moves diagonally. This code is messy, but I think it works.

Near the end of the function is a line that checks if `crsmin==1.01`. This is true only if there is no next box, despite the fact that `pos2box`, in the first line, said that there was one. It can arise from round-off error.

`int reassignmolecs(simptr sim,int diffusing,int reborn);`

Reassigns molecules to boxes. If `diffusing` is 1, only molecules in lists that include diffusing molecules (`sim->mols->diffuselist`) are reassigned; otherwise all lists are reassigned. If `reborn` is 1, only molecules that are reborn, meaning with `indices` greater than or equal to `topl[ll]`, are reassigned; otherwise, entire lists are reassigned. If `reborn` is 1, this assumes that all molecules that are in the system are also in a box, meaning that the `box` element of the molecule structure lists a box and that the `mol` list of that box lists the molecule. Molecules are arranged in boxes according to the location of the `pos` element of the molecules. Molecules outside the set of boxes are assigned to the nearest box. If more molecules belong in a box than actually fit, the number of spaces is doubled using `expandbox`. The function returns 0 unless memory could not be allocated by `expandbox`, in which case it fails and returns 1.

This function was modified on 1/15/16 so that if `reborn` is 1, then molecules are only moved if they are in the wrong places and if so, then they are removed from the old box and placed in the new box, but if `reborn` is 0, then all boxes are cleared out and molecules are assigned from scratch. This is a much faster routine than it was before, especially if boxes have a lot of molecules in them.

## 4.8 Compartments (functions in smolcompart.c)

Compartments are regions of volume that are bounded by surfaces. They do not include their bounding surfaces. They have no function in the performance of the simulation, but are useful for input, output, and communication with MOOSE. Compartments do not maintain a record of what they contain, but instead they define a set of rules that make it is possible to test whether objects are inside or outside of the compartment. Compartments may be disjoint and they may overlap each other.

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. In addition, compartments can be combined with previously defined compartments with logic arguments. Thus, for example, a cell cytoplasm compartment can be defined with the logic equation: equal to the cell compartment and not the nucleus compartment.

### Data structures

```
enum CmptLogic {CLequal,CLequalnot,CLand,CLor,CLxor,CLandnot,CLornot,CLnone};

typedef struct compartstruct {
  struct compartsuperstruct *cmptss;  // compartment superstructure
  char *cname;                 // compart. name (reference, not owned)
  int nsrf;                   // number of bounding surfaces
  surfaceptr *surflist;        // list of bounding surfaces [s]
  int npts;                   // number of inside-defining points
  double **points;             // list of inside-defining points [k][d]
  int ncmptl;                  // number of logic compartments
  struct compartstruct **cmptl; // list of logic compartments [cl]
  enum CmptLogic *clsym;         // compartment logic symbol [cl]
  double volume;               // volume of compartment
  int maxbox;                  // maximum number of boxes in compartment
  int nbox;                   // number of boxes inside compartment
  boxptr *boxlist;             // list of boxes inside compartment [b]
  double *boxfrac;             // fraction of box volume that's inside [b]
  double *cumboxvol;           // cumulative cmpt. volume of boxes [b]
```

```
  } *compartptr;
```

The volume of a compartment is initialized to 0, and is also reset to 0 whenever its definition changes. This indicates that it needs to be updated.

```
typedef struct compartsuperstruct {
  enum StructCond condition;     // structure condition
  struct simstruct *sim;         // simulation structure
  int maxcmpt;                  // maximum number of compartments
  int ncmpt;                    // actual number of compartments
  char **cnames;                // compartment names
  compartptr *cmptlist;         // list of compartments
  } *compartssptr;
```

This structure contains information about all of the compartments. condition is the current condition of the superstructure and sim is a pointer to the simulation structure that owns this superstructure.

**enumerated types**

`enum CmptLogic compartstring2cl(char *string);`
Converts compartment logic symbol string to an enumerated compartment logic type. Input strings can be: “equal”, “equalnot”, “and”, “or”, “xor”, “andnot”, or “ornot”. Anything else results in `CLnone`.

`char *compartcl2string(enum CmptLogic cls,char *string);`
Converts enumerated compartment logic type to a string, in `string`, which must be pre-allocated. Output strings are “equal”, “equalnot”, “and”, “or”, “xor”, “andnot”, “ornot” or “none”. `string` is returned to allow for function nesting.

**low level utilities**

`int posincompart(simptr sim,double *pos,compartptr cmpt,int useoldpos);`
Tests if position `pos` is in compartment `cmpt`, returning 1 if so and 0 if not. This includes composed compartment logic tests. It does not use the compartment box list. This function is quite efficient for surfaces with few panels, but inefficient if surfaces have lots of panels. `useoldpos` tells the function to use the old surface panel position variables `oldpoint` and `oldfront` rather then the current ones.

`int compartrandpos(simptr sim,double *pos,compartptr cmpt);`
Returns a random position, in `pos`, within compartment `cmpt`. Returns 0 and a valid position, unless a point cannot be found, in which case this returns 1.

**memory management**

`compartptr compartalloc(void);`
Allocates memory for a compartment. All arrays are set to `NULL`, and not allocated. Returns the compartment or `NULL` if unable to allocate memory.

`void compartfree(compartptr cmpt);`
Frees a compartment, including all of its arrays.

`compartssptr compartssalloc(compartssptr cmptss,int maxcmpt);`
Allocates a compartment superstructure as well as `maxcmpt` compartments. Space is allocated and initialized for compartment names. Returns the compartment superstructure or `NULL` if unable to allocate memory. This function may be called multiple times in order to space for additional compartments. See `surfacessalloc`.

`void compartssfree(compartssptr cmptss);`
Frees a compartment superstructure, including all compartments and everything within them.

**data structure output**

`void compartoutput(simptr sim);`
Displays all important information about all compartments to stdout.

`void writecomparts(simptr sim,FILE *fptr);`
Prints information about all compartments to file `fptr` using a format that allows the compartments to be read as a configuration file.

`void checkcompartparams(simptr sim);`
This checks a few compartment parameters.

**structure set up**

`void compartsetcondition(compartssptr cmptss,enum StructCond cond,int upgrade);`
Sets the compartment superstructure condition to `cond`, if appropriate. Set `upgrade` to 1 if this is an upgrade, to 0 if this is a downgrade, or to 2 to set the condition independent of its current value. If the condition is downgraded, this also downgrades the simulation structure condition.

`int compartenablecomparts(simptr sim,int maxcmpt);`
Enables compartments in the simulation by allocating the compartment superstructure to hold `maxcmpt` compartments and setting the necessary condition state. This function may be called multiple times. Returns 0 for success or 1 if memory could not be allocated. Function is analogous to `surfenablesurfaces`.

`compartptr compartaddcompart(simptr sim,char *cmptname);`
Adds a compartment named `cmptname` to `sim`. This allocates all necessary memory, including the superstructure if needed. Returns a pointer to the compartment for success or `NULL` for failure. Function is analogous to `surfaddsurface`.

`int compartaddsurf(compartptr cmpt,surfaceptr srf);`
Adds surface `srf` to the compartment `cmpt`. This increments `nsrf` and appends the surface to `srflist`. Returns 0 for success, 1 if memory could not be allocated, and 2 if the surface was already in the list (in which case it is not added again).

`int compartaddpoint(compartptr cmpt,int dim,double *point);`
Adds point `point` to the compartment `cmpt`, in a `dim` dimensional system. This increments `npts` and appends the point to points. Returns 0 for success and 1 if memory could not be allocated.

`int compartaddcmptl(compartptr cmpt,compartptr cmptl,enum CmptLogic sym);`
Add logically composed compartment `cmptl`, which is composed with symbol `sym`, to the compartment `cmpt`. This increments `ncmptl` and appends the new logic compartment to `cmptl`. Returns 0 for success, 1 if memory could not be allocated, or 2 if `cmpt` and `cmptl` are the same, which is not allowed.

`int compartupdatebox(simptr sim,compartptr cmpt,boxptr bptr,double volfrac);`
Updates the listing of box `bptr` in compartment `cmpt`, according to the rule that boxes should be listed if any portion of them is within the compartment and should not be listed if no portion is within the compartment. This also updates the `cumboxvol` and volume structure elements as needed. If the fraction of the box within the compartment is known, including 0, enter it in `volfrac`. If it is unknown and should be calculated, enter -1 in `volfrac`. If the fraction is unknown and should be unchanged if the box was already in the compartment and calculated if the box wasn't in the compartment, then enter -2 in `volfrac`. This returns 0 for no change, 1 for box successfully added, 2 for box successfully removed, 3 for box was already listed but volume was updated, or -1 for failure to allocate memory. If the volume of the box within the compartment needs to be calculated, this calculates it with a hard-coded value of 100 random trial points. Memory is allocated as needed.

The following table lists the return values, which is useful for understanding them and for reading through the function. The former value for each pair is for the actual volume fraction, in `volfrac2`, equal to 0 and the latter is for the actual volume fraction >0.

| | | value of `volfrac` | | |
|---|---|---|---|---|
| | | -2 | -1 | 0 to 1 |
| `bptr` was in `cmpt` | yes | 0 / 0 | 2 / 0,3 | 2 / 0,3 |
| `(bc<=cmpt->nbox)` | no | 0 / 1 | 0 / 1 | 0 / 1 |

`compartptr compartreadstring(simptr sim,compartptr cmpt,char *word,char *line2,char *erstr);`
Reads and processes one line of text from the configuration file, or some other source, for the compartment `cmpt`. If the compartment is not known, then set `cmpt` to `NULL`. The first word of the line should be sent in as `word` and the rest sent in as `line2`. If this function is successful, it returns the compartment and it does not change the contents of `erstr`; if not, it returns `NULL` and writes an error message to `erstr`.

`int loadcompart(simptr sim,ParseFilePtr *pfpptr,line2,char *erstr);`
Loads a compartment, or information for an already existing compartment, from an already opened configuration file. This is used to fill in basic compartment details. However, it does not address any of the box information. Returns 0 for success and 1 for an error; error messages are returned in `erstr`.

`int compartsupdateparams(simptr sim);`
Sets up the boxes and volumes portions of all compartments. Returns 0 for success, 1 for inability to allocate sufficient memory, or 2 for boxes not set up before compartments. This function may be run during initial setup, or at any time afterwards. It is computationally intensive.

`int compartsupdatelists(simptr sim);`
Does nothing. This function is here for future expansion, and to keep similarity between different Smoldyn modules.

`int compartsupdate(simptr sim);`
Sets up or updates all portions of compartment data structures.

**core simulation functions**

`void comparttranslate(simptr sim,compartptr cmpt,int code,double *translate);`
Translates compartment `cmpt` by the displacement given in `translate`. The different bits of `code` tell which attributes of the compartment should be translated. If `code&1`, then the bounding surfaces of the compartment (that are listed in the compartment definition, not including those that are implied through the logic statements) are translated. If `code&2`, then the molecules that are bound to the bounding surfaces of the compartment (which are listed in the compartment definition) are translated. If `code&4`, then the molecules that are inside the compartment are translated. If `code&8`, then molecules that get bumped into by the moving surfaces get translated.

This has a number of flaws with the external molecules. The algorithm is that any molecule that would get bumped into gets translated by the same amount as the compartment, except if its surface action is "transmit" for both surface faces. After being translated, the function checks for its collisions with any surfaces, dealing with them as required. One problem is that this ignores most surface actions, except only for a slight check for transmit vs. anything else. Another problem is that molecules will end up on the wrong side of the moving surface if they get squeezed between the moving surface and a static surface. Here, what happens is the molecule is moved because of the moving surface, then it bounces off of the static surface back towards where it came from, and then the moving surface moves over it. In some ways, this is unavoidable because there is no good solution for what to do with squeezed molecules. However, it seems that it could be improved.

## 4.9 Ports (functions in smolport.c)

Ports are data structures 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 put into the buffer for export. Alternatively, molecules may be added to the Smoldyn simulation at the porting surface by other programs. To perform porting within Smoldyn, a command called transport will move molecules from one port to another. This can be used to test porting, or, hopefully, to transport molecules between multiple Smoldyn simulations.

As much as possible, the code for ports is very analogous to the code for compartments.

## Data structures

```
typedef struct portstruct {
  struct portsuperstruct *portss; // port superstructure
  char *portname;              // port name (reference, not owned)
  surfaceptr *srf;             // porting surface (ref.)
  enum PanelFace face;         // active face of porting surface
  int llport;                  // live list number for buffer
  } *portptr;

typedef struct portsuperstruct {
  enum StructCond condition;   // structure condition
  struct simstruct *sim;       // simulation structure
  int maxport;                 // maximum number of ports
  int nport;                   // actual number of ports
  char **portnames;            // port names
  portptr *portlist;           // list of ports
  } *portssptr;
```

This structure contains information about all ports. `condition` is the current condition of the superstructure and `sim` is a pointer to the simulation structure that owns this superstructure.

**memory management**

`portptr portalloc(void);`
Allocates memory for a port. Pointers are set to `NULL` and llport is set to -1. Returns the port or `NULL` if unable to allocate memory.

`void portfree(portptr port);`
Frees a port.

`portssptr portssalloc(portssptr portss,int maxport)`
Allocates a port superstructure, if needed, as well as `maxport` ports. Space is allocated and initialized for port names and port lists. Returns the port superstructure or `NULL` if unable to allocate memory. This function can be called repeatedly to expand the number of ports.

`void portssfree(portssptr portss);`
Frees a port superstructure, including all ports.

**data structure output**

`void portoutput(simptr sim);`
Displays all important information about all ports to stdout.

`void writeports(simptr sim,FILE *fptr);`
Prints information about all ports to file `fptr` using a format that allows the ports to read as a configuration file.

`int checkportparams(simptr sim,int *warnptr);`
This checks a few port parameters.

**structure set up**

`void portsetcondition(portssptr portss,enum StructCond cond,int upgrade);`
Sets the port superstructure condition to `cond`, if appropriate. Set `upgrade` to 1 if this is an upgrade, to 0 if this is a downgrade, or to 2 to set the condition independent of its current value. If the condition is downgraded, this also downgrades the simulation structure condition.

`int portenableports(simptr sim,int maxport);`
Enables ports in the simulation by allocating the port superstructure to hold `maxport` ports and setting the necessary condition state. This function may be called multiple times. Returns 0 for success or 1 if memory could not be allocated. Function is analogous to `surfenablesurfaces`.

`portptr portaddport(simptr sim,char *portname,surfaceptr srf,enum PanelFace face);`
Adds, or updates, a port to the port superstructure. If `portname` is not the name of an existing port, a new port is defined, the `nport` element of the superstructure is incremented, and this name is copied over for the new port. Alternatively, if `portname` has already been defined, it is used to reference an existing port. Either way, the port surface is set to `srf` if `srf` is not `NULL`, the port face is set to `face` if `face` is not `PFnone`, and the address of the port is returned.

`portptr portreadstring(simptr sim,portptr port,char *word,char *line2,char *erstr);`
Reads and processes one line of text from the configuration file, or some other source, for the port `port`. If the port is not known, set `port` to `NULL`. The first word of the line should be sent in as `word` and the rest sent in as `line2`. If this function is successful, it returns the port and it does not change the contents of `erstr`; if not, it returns `NULL` and it writes an error message to `erstr`.

`int loadport(simptr sim,ParseFilePtr *pfpptr,char* line2,char *erstr);`
Loads a port, or information for an already existing port, from an already opened configuration file. This is used to fill in basic port details. However, it does not assign a molecule buffer to the port. Returns 0 for success and 1 for an error; error messages are returned in `erstr`.

`int portsupdateparams(simptr sim);`
Does nothing. This function is here for future expansion, and to maintain similarity between different Smoldyn modules.

`int portsupdatelists(simptr sim);`
Sets up the molecule buffers for all ports. Returns 0 for success, 1 for inability to allocate sufficient memory, or 2 for molecules not set up sufficiently.

`int portsupdate(simptr sim);`
Sets up or updates all port data structure components.

**core simulation functions**

`int portgetmols(simptr sim,portptr port,int ident,enum MolecState ms,int remove);`
Returns the number of molecules of type `ident` (use -1 for all species) and state `ms` (`MSall` is allowed) that are in the export buffer of port `port`. If `remove` is 1, this kills those molecules, so that they will be returned to the dead list at the next sorting; otherwise they are left in the port. The intention is that molecules that are gotten from the export list with this function are then added to MOOSE or another simulator.

`int portputmols(simptr sim,portptr port,int nmol,int ident,int *species,double **positions,double **positionsx);`
Adds `nmol` molecules of state `MSsoln` to the simulation system at the porting surface of port `port`. If `species` is non-`NULL`, then it needs to be an `nmol` length list of species numbers, which are used for the molecule species and `ident` is ignored; alternatively, if `species` is `NULL`, then all molecules will have species `ident`. Likewise, if `positions` is non-`NULL`, then it needs to be a list of molecule positions that will be used for the new molecule positions. If `positions` is `NULL`, then molecules are placed randomly on the porting surface. If `positionsx` is `NULL`, then molecule old positions are fixed to the panel, and otherwise molecule old positions are set to these values. This returns 0 for success, 1 for insufficient available molecules, 2 for no porting surface defined, 3 for no porting surface face defined, and 4 for no panels on porting surface.

`int porttransport(simptr sim1,portptr port1,simptr sim2,portptr port2);`
Transports molecules from `port1` of simulation structure `sim1` to `port2` of simulation structure `sim2`. `sim1` and `sim2` may be the same and `port1` and `port2` may be the same. This is designed for testing ports or for coupled Smoldyn simulations that communicate with ports.

## 4.10 Lattices (functions in smollattice.c)

A lattice is a region of space in which molecules do not have precise spatial locations, but are compartmentalized to lattice subvolumes. At present, lattices use axis-aligned rectangular subvolumes, but they could at some point be unstructured meshes (as in URDME). In the lattice region of space, molecules are currently represented as discrete individuals, in each subvolume. At some point though, this could be extended to continuous concentrations, to enable PDE simulation methods. Lattice regions of space interface to particle-based regions of space through Smoldyn's ports.

Martin Robinson added lattice functionality to Smoldyn during 2013. This documentation is my best understanding of his code, which I don't promise is correct. The lattice code is in several subdirectories of `source`. The `NextSubVolume` directory contains Martin's code for the next subvolume simulation method. The `vtk` directory contains a couple of VTK wrapper code files, which I assume are used to interface Martin's code to the VTK visualization software. I suspect that Martin wrote those wrapper files. Finally, the lattice code is integrated into the rest of Smoldyn using the smollattice.c file, which compiles with the rest of Smoldyn. The code in smollattice.c is always compiled, regardless of compiling options, but if the `LATTICE` configuration variable is undefined, then all calls to the nsv code are disabled.

### Data structures

```
enum LatticeType {LATTICEnone,LATTICEnsv,LATTICEpde};

typedef struct latticestruct {
  struct latticesuperstruct *latticess; // lattice superstructure
  char *latticename;          // lattice name (reference, not owned)
  enum LatticeType type;      // type of lattice
  double min[DIMMAX];         // lower spatial boundaries
  double max[DIMMAX];         // upper spatial boundaries
  double dx[DIMMAX];          // lattice lengthscale (subvolume width)
  char btype[DIMMAX];         // boundary type (r)eflective or (p)eriodic
  portptr port;               // interface port (ref.)
  int **convert;    // convert to particle at port, 0 or 1 [lat.species][face] ??
  int maxreactions;           // maximum number of reactions
  int nreactions;             // number of reactions
  rxnptr *reactionlist;       // list of reactions
  int *reactionmove;          // 0 or 1 for moving reactions
  maxsurfaces??
  nsurfaces??
  surfacelist??
  int maxspecies;             // maximum number of species
  int nspecies;               // number of species
  int *species_index;         // species indices
  int *maxmols;               // allocated size of molecule list [lat.species]
  int *nmols;                 // number of individual molecules [lat.species]
  double*** mol_positions;    // molecule positions [lat.species][nmols][dim]
  NextSubvolumeMethod* nsv;   // nsv class
  NextSubvolumeMethod* pde;   // pde class
  } *latticeptr;

typedef struct latticesuperstruct {
  enum StructCond condition;  // structure condition
  struct simstruct *sim;      // simulation structure
  int maxlattice;             // maximum number of lattices
  int nlattice;               // actual number of lattices
  char **latticenames;        // lattice names
  latticeptr *latticelist;    // list of lattices
  } *latticessptr;
```

These structures contain information about all lattices. Starting with the lattice superstructure, `condition` is the current condition of the superstructure and `sim` is a pointer to the simulation structure that owns this superstructure. These, and other superstructure elements are completely standard. Space is allocated for `maxlattice` lattices, of which `nlattice` are actually defined and used. Their names are in `latticenames` and they are pointed to by pointers in `latticelist`.

In the lattice structure, of type `latticestruct`, the `latticess` pointer points to the owning superstructure and `latticename` points to the lattice name. Each lattice is either type `LATTICEnsv` if it is for discrete particles or `LATTICEpde` if it is for continuous concentrations. There is also the enumeration `LATTICEnone`, which is helpful for specifying neither of the other options but is not valid for actual lattices. `min` and `max` vectors represent the lower and upper spatial boundaries of the lattice space; this space is divided into subvolumes that each have width given with the `dx` vector. `btype` is a vector with a character for each coordinate that equals 'r' for reflective boundaries, 'p' for periodic boundaries, and 'u' for undefined.

The `port` element points to the port that divides the lattice part of space from the particle-based part of space. The `maxreactions` and `nreactions` elements tells how many reactions are allocated and are being used here. These reactions, listed in `reactionlist` are simply pointers to Smoldyn's normal reactions. This means that there is no need for additional reaction data structures here. `reactionmove` is either 0 for 1 for each reaction, where 1 means that it should be moved and 0 means that it should not be moved; a moved reaction is only implemented on the lattice side of space and is set to a rate of 0 on the particle side of space. `nspecies` is the number of species that are declared for the lattice region of space. It may be different from `nspecies` values in the particle side of space, defined in the rest of Smoldyn. `species_index` is a list of species indices from the particle side of space; it is a look-up table that connects lattice species numbers to particle side species numbers. `maxmols` is the allocated size of the molecule list and `nmols` is the number of molecules for each species, indexed with the lattice-side indices. `mol_positions` are the positions of the molecules, indexed as the lattice-side species number, molecule number, and dimensional coordinate. I don't really understand this because I didn't think these molecules had precise positions. The `nsv` and `pde` pointers point to classes in the core NSV and PDE code (the PDE code isn't included yet).

## Functions

**memory management**

`latticeptr latticealloc(int dim);`
: Allocates memory for a lattice assuming a `dim` dimensional system. Pointers are set to `NULL` and values are set to 0 or other defaults. Returns the lattice or `NULL` if unable to allocate memory.

`void latticefree(latticeptr lattice);`
: Frees a lattice and all of the memory in it.

`int latticeexpandreactions(latticeptr lattice,int maxrxns);`
: Expands the number of reactions in a lattice to `maxrxns`, allocating new memory and freeing old memory as needed. This addresses the memory in `reactionlist` and `reactionmove`. Returns 0 for success or 1 for failure.

`int latticeexpandspecies(latticeptr lattice,int maxspecies);`
: Expands the number of species in a lattice to `maxspecies`, allocating new memory and freeing old memory as needed. This addresses the memory in `species_index` and `nmols`. It also addresses the memory in `mol_positions`. Returns 0 for success or 1 for failure.

`int latticeexpandmols(latticeptr lattice,int species,int maxmols,int dim);`
: Expands the `mol_positions` list for the lattice species number `species` so that it will have `maxmols` spaces in it, each with `dim` values.

`latticessptr latticessalloc(latticessptr latticess,int maxlattice)`
: Allocates a lattice superstructure, if needed, as well as `maxlattice` lattices. Space is allocated and initialized for lattice names and lattice lists. Returns the lattice superstructure or `NULL` if unable to allocate memory. This function can be called repeatedly to expand the number of lattices.

`void latticessfree(latticessptr latticess);`
Free a lattice superstructure and all of its contents.

**data structure output**

`void latticeoutput(simptr sim);`
Outputs the contents of the lattice superstructure and all of the member lattices to the display. This shows essentially everything except for the molecule positions.

`void writelattices(simptr sim,FILE *fptr);`
Writes the contents of the lattice superstructure and all of the member lattices to file `fptr` in a format that enables it to be read in again.

`int checklatticeparams(simptr sim,int *warnptr);`
Checks the lattice parameters, sending output to the simLog function along with the appropriate error level.

**structure set up**

`void latticesetcondition(latticessptr latticess,enum StructCond cond,int upgrade);`
Sets the lattice superstructure condition to `cond`, if appropriate. Set `upgrade` to 1 if this is an upgrade, to 0 if this is a downgrade, or to 2 to set the condition independent of its current value. If the condition is downgraded, this also downgrades the simulation structure condition.

`int latticeenablelattices(simptr sim);`
Enables simulation with lattices. This function may be called more than once but doesn't do anything on repeat calls. This allocates the superstucture and 1 lattice. It also initializes the NSV code.

`latticeptr latticeaddlattice(simptr sim,latticeptr *latptr,const char *latticename,const double *min,const double *max,const double *dx,const char *btype,enum LatticeType type);`
Adds a new lattice or modifies an existing lattice. This enables lattices, creates the superstructure, and allocates memory as needed. Send in `latptr` with a pointer to the lattice if it has already been created and now needs to be modified, or `NULL` if a new lattice is wanted. In the latter case, this pointer is returned with the address of the new lattice. All of the lattice parameters that can be set here are optional, meaning that `NULL` can be entered to choose to not set the parameter. If entered, the lattice name is set to `latticename`, the minimum boundary coordinates to `min`, the maximum boundary coordinates to `max`, the lattice length parameters to `dx`, the boundary types to `btype`, and the lattice type to `type`. In the last case, enter `LATTICEnone` to not set this parameter. Returns 0 for success or 1 for inability to allocate memory.

`int latticeaddspecies(latticeptr lattice,int ident,int *index);`
Adds one or more Smoldyn species to an existing lattice, allocating new memory as needed. If `index` is `NULL`, this sets the `species_index` element of the lattice structure to the `ident` value that is entered (the index of the Smoldyn species) and this also sets the number of molecules of this species on the lattice to 0. Returns 0 for success, 1 for inability to allocate memory, or 2 if this species was already in the lattice. If `index` is not `NULL`, then this adds each of the listed species to the lattice, returning 0 for success or 1 for inability to allocate memory; in this case any duplicate species are just ignored.

`int latticeaddrxn(latticeptr lattice,rxnptr reaction,int move);`
Adds a reaction to an existing lattice, allocating new memory as needed. This puts the reaction at the end of the current reaction list and sets its "move" status to the `move` value that is entered. Returns 0 for success, 1 for inability to allocate memory, or 2 if the reaction was already in the lattice.

`int latticeaddmols(latticeptr lattice,int nmol,int i,double *poslo,double *poshi,int dim);`
Adds `nmol` molecules of Smoldyn species `i` to lattice `lattice`. These molecules are randomly postioned in the axis-aligned bounding box defined by the `poslo` and `poshi` vectors. The space has `dim`

dimensions. This allocates memory as needed. This also adds the species index `i` to the lattice if it's not already there. Returns 0 for success or 1 for failure to allocate memory.

`void latticeaddport(latticeptr lattice,portptr port);`
Sets the port for a lattice. A lattice only works with a single port, so there's no need to allocate memory, check for prior status, etc. Instead, this simply sets the port variable.

`int latticeaddconvert(latticeptr lattice,int ident,int *index,enum PanelFace face,int convert);`
Sets the lattice `convert` element for the single Smoldyn species `ident`, or for the group of Smoldyn species in `index`, and the face `face` to the value in `convert`. This is for lattice-molecules that collide with the given face side of the port, to indicate if they should convert to particle molecules or stay as lattice molecules. Returns 0 for success or the identity of the Smoldyn species if the Smoldyn species has not been imported to the lattice.

`latticeptr latticereadstring(simptr sim,ParseFilePtr pfp,latticeptr lattice,const char *word,char *line2);` This reads and processes user input, much like `portreadstring` and similarly named functions. As always, `sim` is the simulation pointer, `pfp` is the parse file pointer, which contains the file information, `lattice` is the pointer to the current lattice being worked on or `NULL` if there is no current lattice, `word` is the first word of the input string, and `line2` is the remainder of the input string. This returns a pointer to the current lattice being worked on if successful and `NULL` if there is an error; in the latter case, the error is first sent to `simParseError`.

`int loadlattice(simptr sim,ParseFilePtr *pfpptr,char* line2);` Loads a lattice using input from the user. This function is called by `loadsim` when file parsing reaches "start_lattice". This is essentially identical in structure to `loadport`.

`int latticesupdateparams(simptr sim);`
Updates the parameters of all lattices, meaning that it sends various parameters from the lattice structures to the NSV code. This sends the port details, including the porting rectangle panel locations, dimensions, and orientations. This also sends the list of reactions to the NSV code. Returns 0 for success or 1 for inability to allocate memory.

`int latticesupdatelists(simptr sim);`
Updates the more fundamental parameters of all lattices, meaning that it sends them from the lattice structures to the NSV code. First, this deletes any existing nsv data. Then, this sends the lattice dimensions, the species, and the numbers of molecules of each species. This deletes the local molecule data afterwards.

`int latticesupdate(simptr sim);`
Updates the lattice superstructure and all member lattices as needed, while updating the condition as appropriate. This function is essentially identical to ones with similar names in other code modules.

**core simulation functions**

`int latticeruntimestep(simptr sim);`
Runs the lattice NSV code for one simulation time step for each lattice. Returns 0.

**NSV functions, in nsvc.cpp**

`void nsv_init();`
Initializes the Kairos simulation engine.

`NextSubvolumeMethod* nsv_new(double* min, double* max, double* dx, int n);`
Allocates and initializes a new nsv type data structure, and then returns it. `min` and `max` are the low and high corners of space, `dx` is the lattice spacing on each coordinate, and `n` is the lattice dimensionality.

`void nsv_delete(NextSubvolumeMethod* nsv);`
Deletes an nsv type data structure.

`void nsv_print(NextSubvolumeMethod* nsv, char** bufferptr);`
Prints out information about the `nsv` data structure to the buffer, which is simply a pointer to an unallocated string. The buffer needs to be freed afterwards.

`void nsv_add_interface(NextSubvolumeMethod* nsv,int id,double dt, double *start,double *end,double *norm,int dim)`
Adds information about a porting surface panel to the lattice `nsv`. This panel has to be a rectangle shape. It is used for conversion of a particular species at this surface. `id` is the Smoldyn species index (not the lattice species index) of the species that should be ported at the interface. `dt` is the Smoldyn time step (same for the lattice). `start` is a `dim`-dimensional list of the rectangle low-value coordinates and `end` is a list of the rectangle high-value coordinates. `norm` is the `dim`-dimensional surface normal vector that points towards the particle side of the interface and away from the lattice-side of the interface. Finally, `dim` is the simulation dimensionality.

`void nsv_add_species(NextSubvolumeMethod* nsv,int id,double D,char *btype,int dim);`
Adds a new species to the lattice `nsv`. `id` is the Smoldyn species number and `D` is the solution-phase diffusion coefficient of this species. `btype` is a vector with the boundary type on each axis, where the options are 'r' for reflective and 'p' for periodic. `dim` is the lattice dimensionality.

`extern void nsv_add_surface(NextSubvolumeMethod* nsv,surfacestruct* surface);`
Adds a surface to the lattice `nsv`. `surface` is a pointer to a Smoldyn surface that should be added to the lattice.

`void nsv_add_reaction(NextSubvolumeMethod* nsv,rxnstruct *reaction);`
Adds a reaction to the lattice `nsv`. `reaction` is a pointer to a Smoldyn reaction that should be added to the lattice.

`void nsv_integrate(NextSubvolumeMethod* nsv,double dt, portstruct *port, latticestruct *lattice);`

Runs the lattice simulation over `dt` amount of time. `nsv` is the lattice, `port` is the port that is between the lattice and the Smoldyn simulation, and `lattice` is the Smoldyn lattice data structure.

`void nsv_add_mol(NextSubvolumeMethod* nsv,int id, double* pos, int dim);`
Adds one molecule to lattice `nsv`. The molecule has Smoldyn species `id` and `dim`-dimensional location vector `pos`. `dim` is the lattice dimensionality.

`int nsv_get_species_copy_numbers(NextSubvolumeMethod* nsv, int id,const int **copy_numbers, const double** positions);`
Gets the copy number of a species in the `nsv` data structure. Enter `id` as the Smoldyn species number (not the lattice species number), `copy_numbers` as a pointer to an unallocated integer array and `positions` as an unallocated double array. Both of these need to freed afterwards.

## 4.11 Filaments (functions in smolfilament.c)

Filament support is in progress.

### Data structures declared in smoldyn.h

```
enum DynamicsType {DTnone,DTrouse,DTalberts};
```

The `DynamicsType` enumerated type describes the filament simulation dynamics type.

```
typedef struct beadstruct {
  double xyz[3];              // bead coordinates
  double xyzold[3];           // bead coordinates for prior time
  } *beadptr;
```

A filament can have beads or segments, but not both. In the bead case, each bead is quite simple. A bead has its current position, in `xyz` and its old position, in `xyzold`. For 2D simulations, only the first two vales are used in each position vector.

```
typedef struct segmentstruct {
  double xyzfront[3];          // Coords. for segment front
  double xyzback[3];           // Coords. for segment back
  double len;                  // segment length
  double ypr[3];               // relative ypr angles
  double dcm[9];               // relative dcm
  double adcm[9];              // absolute segment orientation
  double thk;                  // thickness of segment
  } *segmentptr;
```

Segments are a bit more complicated than beads. `xyzfront` is the $x, y, z$ coordinate of the segment starting point and `xyzback` is the $x, y, z$ coordinate of the segment end point. `len` is the segment length. `ypr` is the relative yaw, pitch, and roll angle for the segment relative to the orientation of the prior segment. `sdcm` is the relative direction cosine matrix for the segment relative to the orientation of the one before it. Identical information is contained in `ypr` and `dcm`, but the latter is here for faster computation. `adcm` is the direction cosine matrix for the absolute orientation. `thk` is the thickness of the segment. For 2D simulations, only the first 2 values are used in the coordinates vectors and only the first value is used in the ypr angles.

At present, filaments are designed for all segments of the same length. This wasn't what I had in mind initially, but may be sufficiently easier to program that it's worth keeping this constraint.

```
typedef struct filamenttypestruct {
  struct filamentsuperstruct *filss;  // filament superstructure
  char *ftname;                // filament type name
  double color[4];             // filament color
  double edgepts;              // thickness of edge for drawing
  unsigned int edgestipple[2];  // edge stippling [factor, pattern]
  enum DrawMode drawmode;      // polygon drawing mode
  double shiny;                // shininess
  enum DynamicsType dynamics; // Dynamics for the filament
  int isbead;                  // 1 for bead model, 0 for segment model
  double stdlen;               // minimum energy segment length
  double stdypr[3];            // minimum energy bend angle
  double klen;                 // force constant for length
  double kypr[3];              // force constant for angle
  double kT;                   // thermodynamic temperature, [0,inf)
  double treadrate;            // treadmilling rate constant
  double viscosity;            // viscosity
  double beadradius;           // bead radiues
  } *filamenttypeptr;
```

Filament types describe various types of filaments, such as actin, microtubule, etc. All filaments of the same type have the same graphical display, force constants, dynamic simulation methods, etc.

Starting with graphical display parameters, `color` is the filament color, with red, green, blue, and alpha values. `edgepts` is the edge thickness for drawing, `edgestipple` is the stippling code for the edge stippleing, if any. `drawmode` is the polymer drawing mode, which can be face, edge, vertex, of a combination of these. `shiny` is the shininess for graphical display. These graphical display statements are very similar to those used for surfaces.

Filament mechanics information is in the next elements. `dynamics` gives the type of simulation dynamics. `stdlen` is the standard segment length, meaning the segment length at minimum energy, where segments are neither stretched nor compressed. `stdypr` is the minimum energy relative ypr angle. `klen` is the stretching force constant. Its value is $< 0$ for an infinite force constant, meaning that the segment lengths are fixed. `kypr` is the bending force constant. For any element, its value is 0 for zero force constant, meaning a freely jointed chain, and $< 0$ for an infinite force constant, meaning a fixed bending angle. `kT` is the thermodynamic energy. This parameter isn't ideal here because it allows a different temperature for different components in the simulation, but it's here for now. `treadrate` is the treadmilling rate constant. `viscosity` is the

medium viscosity for computing drag coefficients. `beadradius` is the radius of beads, which only applies to bead models. For 2D filaments, `stdypr` values 1 and 2 should have value 0 and `kypr` values 1 and 2 should have value -1 (i.e. bending out of the $x, y$ plane can't happen).

```
typedef struct filamentstruct {
  struct filamentsuperstruct *filss;  // filament superstructure
  filamenttypeptr filtype;  // filament type structure
  char *fname;                // filament name
  int maxseg;                 // number of segments allocated
  int nseg;                   // number of segments
  int front;                  // front index
  int back;                   // back index
  beadptr *beads;           // array of the beads ?? Initialise to NULL
  segmentptr *segments;       // array of the segments ?? Initialise to NULL
  } *filamentptr;
```

Each filament is a pretty simple data structure, including only a type and a list of either beads or segments. Beads or segments are joined end-to-end. Multiple filaments of the same type (e.g. two microtubules) are stored in multiple data structures. At some point, we'll add support for filament joining, such as for multi-scale filaments, branched filaments, filament meshes, etc., but that's not here now.

The `filss` element points to the filament superstructure that owns this filament. `fname` is the name of the filament. The filament segment lists are allocated for `maxseg` total segments, of which `nseg` of those are actually used for segments. Each segment list starts at position `front` and continues to position `back-1`. These are implemented so that `back` is always greater than `front`. I considered using a circular queue, but this is easier to use and runs faster in most algorithms.

```
typedef struct filamentsuperstruct {
  enum StructCond condition;  // structure condition
  struct simstruct *sim;      // simulation structure
  int maxfil;                 // maximum number of filaments
  int nfil;                   // actual number of filaments
  char **fnames;              // filament names
  filamentptr *fillist;       // list of filaments
  } *filamentssptr;
```

The superstructure contains a list of filaments. It also contains a condition element and a pointer up the heirarchy to the simulation structure.

## Filament math

Following are the basic equations for the filament relative angles (**A**, `dcm`), absolute angles (**B**, `adcm`), and positions (**x**, `xyz`).

$\mathbf{B}_0 = \mathbf{A}_0$
$\mathbf{B}_i = \mathbf{A}_i \cdot \mathbf{B}_{i-1}$
$\mathbf{B}_i = \mathbf{A}_{i+1}^T \cdot \mathbf{B}_{i+1}$

## Function declarations.

As much as possible, functions are declared locally rather than in the smoldynfuncs.h header file. This simplifies the code reading because it clarifies which functions might be called externally versus those that are only called internally.

`char *fildt2string(enum DynamicsType dt,char *string);` Local. Converts filament dynamics type to string. Returns "none" for unrecognized dynamics type. Writes result to `string` and also returns it directly.

`enum DynamicsType filstring2dt(char *string);` Local. Converts filament dynamics string to enumerated dynamics type. Returns `DTnone` for unrecognized input.

**low level utilities**

`double filRandomLength(const filamenttypeptr filtype,double thickness,double sigmamult);`

Local. Returns a random segment length using the mechanics parameters given in filament type `filtype`. `thickness` is the thickness of the new segment and `sigmamult` is multiplied by the normal standard deviation of the length. The returned length is Gaussian distributed, with mean equal to `fil->lstd` and standard deviation equal to $\sigma_{mult}\sqrt{kT/(thickness * k_{len})}$. The returned length is always positive.

`double *filRandomAngle(filamenttypeptr filtype,double thickness,double *angle,double sigmamult);`

Local. Returns a random bending angle in `angle` (a relative ypr angle) for filament type `filtype`, without changing the filament. The new segment has thickness `thickness` and the normal standard deviation is multiplied by `sigmamult`.

`double filStretchEnergy(filamentptr fil,int seg1,int seg2);`

Computes the stretching energy for filament `fil`. Enter `seg1` as the first segment to compute from, or as -1 to indicate that it should start at the front of the filament, and `set2` as the last segment to compute to, or as -1 to indicate that calculation should end at the end of the filament. The equation is

$$E_{stretch} = \sum_{s} \frac{thk_s k_{len}(l_s - l_{std})^2}{2} \tag{4.1}$$

`double filBendEnergy(filamentptr fil,int seg1,int seg2);`

Computes the bending energy for filament `fil`. Enter `seg1` as the first segment to compute from, or as -1 to indicate that it should start at the front of the filament, and `set2` as the last segment to compute to, or as -1 to indicate that calculation should end at the end of the filament. The equation is

$$E_{bend} = \sum_{s=1}^{n} \frac{thk_{s-1} + thk_s}{2} \frac{k_y(a_y - a_{y,std})^2 + k_p(a_p - a_{p,std})^2 + k_r(a_r - a_{r,std})^2}{2} \tag{4.2}$$

The first term in the sum computes the average thickness of the segment in front of and behind the bend. The other term is the squared bending angle on each coordintate.

`void filArrayShift(filamentptr fil,int shift);`

Shifts the bead or segment list in the filament (beads if they are defined and segements otherwise) either to higher indices or to lower indices, adjusting the `front` and `back` elements to account for the shift. Enter `shift` as a positive number to increase all of the indices, as a negative number to decrease all of the indices, and as 0 to have the filament centered in the allocated memory.

This is inefficient currently since it does repeated memory swaps rather than moving blocks of memory around more intelligently. As a results, empty beads/segments may get moved multiple times in a single array shift.

**Memory management**

`beadptr beadalloc();`

Allocates memory for a single bead and initializes this bead. Returns a pointer to this bead, or `NULL` if memory could not be allocated.

`void beadfree(beadptr bead);`

Frees memory for a single bead.

`segmentptr segmentalloc();`

Allocates memory for a single segment and initializes this segment. Returns a pointer to this segment, or `NULL` if memory could not be allocated.

`void segmentfree(segmentptr segment);`

Frees memory for a single segment.

`filamenttypeptr filamenttypealloc();`
Allocates memory for a filament type structure and intilizes it. Returns a pointer to this filament type, or `NULL` if memory could not be allocated.

`void filamenttypefree(filamenttypeptr filtype);`
Frees memory for a single filament type.

`filamentptr filalloc(filamentptr fil,int maxbead,int maxseg);`
Allocates and initializes a filament for `maxseg` segments and returns the resulting pointer.

`void filfree(filamentptr fil);`
Frees a filament and all of the data structures in it.

`filamentssptr filssalloc(filamentssptr filss,int maxfil);`
Allocates and initializes a filament superstructure for a maximum size of `maxfil` filaments. This is set up for expanding the list, but nevertheless needs work, due to the static memory allocation for individual filaments.

`void filssfree(filamentssptr filss);`
Fres a filament superstructure and all of the structures within it.

**Data structure output**

`void filtypeoutput(filamenttypeptr filtype,int dim);` Outputs all of the key information about a filament type to the display.

`void filoutput(filamentptr fil);`
Outputs all of the key information about a filament to the display.

`void filssoutput(simptr sim);`
Outputs all of the key information about a filament superstructure, and all of the filaments in it, to the display.

`void filwrite(simptr sim,FILE *fptr);`
Writes filament information to a file in Smoldyn format for loading in again later on. This function isn't written yet.

`int filcheckparams(simptr sim,int *warnptr);`
Checks filament parameters to make sure they are all reasonable. This function isn't written yet.

**Filament manipulation**

`int filAddSegment(filamentptr fil,double *x,double length,double *angle,double thickness,char endchar);`
Adds a segment to filament `fil`. If this is the first segment, then `x` needs to be set to the starting location of the filament; otherwise, `x` is ignored. `length` is the length of the segment, `angle` is the relative angle of the segment facing along the filament from front to back (for the first segment, this is the absolute angle), `thickness` is the segment thickness, and `endchar` should be set to 'b' to add to the back of the filament or 'f' to add to the front of the filament. For segments added to the front, `angle` is the new angle from the new first monomer to the next monomer, facing towards the back. If the first segment is added to the back, then `angle` is the angle of the new segment off of the coordinate system. If the first segment is added to the front, then `angle` is the angle from the new segment to the coordinate system.

For the first segment: $\mathbf{x}_0 = \mathtt{x}$, $\mathbf{x}_1 = \mathbf{x}_0 + l_0 \mathbf{B}_0^T \cdot \hat{\mathbf{x}}$

For segments added to the back, with index $i$: $\mathbf{a}_i = \mathtt{angle}$, $\mathbf{A}_i = DCM(\mathbf{a}_i)$, $\mathbf{B}_i = \mathbf{A}_i \cdot \mathbf{B}_{i-1}$, $\mathbf{x}_{i+1} = \mathbf{x}_i + l_i \mathbf{B}_i^T \cdot \hat{\mathbf{x}}$

For segments added to the front, with index $i$ (typically equal to 0): $\mathbf{B}_i = \mathbf{A}_{i+1}^T \cdot \mathbf{B}_{i+1}$, $\mathbf{A}_i = \mathbf{B}_i$ $\mathbf{a}_i = XYZ(\mathbf{B}_i)$, $\mathbf{x}_i = \mathbf{x}_{i+1} - l_i \mathbf{B}_i^T \cdot \hat{\mathbf{x}}$

`int filRemoveSegment(filamentptr fil,char endchar);`
Removes one segment from either the front or back end of a filament. Specify the end in `endchar` with 'f' for front and 'b' for back.

`void filTranslate(filamentptr fil,const double *vect,char func)`
Translates an entire filament. Enter `vect` with a 3-D vector and `func` with '=' for translate to the given posiition, with '-' for to subtract the value of `vect` from the current position, and with '+' to add the value of `vect` to the current position.

`void filRotateVertex(filamentptr fil,int seg,double *angle,char endchar,char func);`
Not written yet. This function is supposed to rotate part of the filament about one of the filament vertices by ypr angle `angle`. The character inputs give which end moves and whether it moves the angle to the given value, whether it subtracts the given value from the current value, or whether it adds the current value.

`void filLengthenSegment(filamentptr fil,int seg,double length,char endchar,char func);`
Not written yet. This function is supposed to modify the length of a single segment. As before, the character inputs give which end moves and whether it moves the length to the given value, whether it subtracts the given value from the current value, or whether it adds the current value.

`void filReverseFilament(filamentptr fil);`
Not written yet. This will reverse the sequence of segments in a filament.

`int filCopyFilament(filamentptr filfrom,filamentptr filto,const char *fname);`
This copies all of the values in a filament to another filament.

**Structure set up**

`void filsetcondition(filamentssptr filss,enum StructCond cond,int upgrade);`
Local. This function sets the condition of the filament superstructure. Set `upgrade` to 1 if this is an upgrade, to 0 if this is a downgrade, or to 2 to set the condition independent of its current value.

`int filSetParam(filamentptr fil,const char *param,int index,double value);`
Sets various parameters for a filament, where the parameter name is `param` and it is set to the value `value`. If this parameter has multiple inidices, enter the index to be changed in `index`, or enter `index` as -1 to set all of the indices at once to the same value. Returns 0 for success or 2 for out of bounds value. The parameter options are: `stdlen` for the standard length, `stdypr` for the standard yaw-pitch-roll angles, `klen` for the stretching force constant, `kypr` for the bending force constants, `kT` for the thermodynamic energy, `treadrate` for the treadmilling rate, `viscosity` for the medium viscosity, and `beadradius` for the bead radius.

`int filsetcolor(filamentptr fil,double *rgba);`
Set the 4-value color vector of the filament to the values entered in `rgba`. Returns 0 unless one of the entered values is outside of the range [0,1], in which case this does not change the current color and returns an error code of 2.

`int filsetedgepts(filamentptr fil,double value);`
Sets the drawing thickness of the filament to the value entered in `value`. Returns 0 for success or 2 if the entered value is less than 0, which is an error.

`int filsetstipple(filamentptr fil,int factor,int pattern);`
Sets the stippling factor and pattern for the filament to the entered values. Returns 0 for success or 2 if values are out of bounds.

`int filsetdrawmode(filamentptr fil,enum DrawMode dm);`
Sets the filament drawing mode to the entered value. Returns 0 for success or 2 if the input value is unrecognized.

`int filsetshiny(filamentptr fil,double shiny);`
Sets the shininess for the filament to the entered value. Returns 0 for success or 2 if the value is out of bounds.

`int filenablefilaments(simptr sim,int maxfil);`
Allocates a filament superstructure with `maxfil` filaments, adds it to the simulation structure, and sets the filament condition to `SClists`. This function can be called multiple times. Use this function to expand the number of filaments in the superstructure.

`filamentptr filaddfilament(simptr sim,const char *fnames);`
Add a new filament to the simulation, which is named `fnames`, returning a pointer to the new filament. If a filament with this name already exists, this returns a pointer to the existing filament.

`int filAddRandomSegments(filamentptr fil,int number,const char *xstr,const char *ystr,const char *zstr,double thickness);`
Adds `number` segments to the existing filament in `fil`. The `xstr`, `ystr`, and `zstr` entries are only considered if the filament previously had no segments; in this case, the filament starting point is at the coordinates given by them. Each entry can be a 'u' to indicate uniform distributed random location in the simulation volume, or a value for the actual coordinate. Enter `thickness` with the segment thickness.

`filamentptr filreadstring(simptr sim,ParseFilePtr pfp,filamentptr fil,const char *word,char *line2);`

Reads one line of user input within a filament block.

`int filload(simptr sim,ParseFilePtr *pfpptr,char *line2);`
Reads multiple lines of user input from within a filament block, calling `filreadstring` with each line.

`int filupdateparams(simptr sim);`
Does nothing currently. This function will take compute any simulation parameters necessary from user parameters.

`int filupdatelists(simptr sim);`
Does nothing currently. This function will take compute any simulation list changes necessary from user parameters.

`int filsupdate(simptr sim);`

**Core simulation functions**

`int filMonomerXSurface(simptr sim,filamentptr fil,char endchar);`

`int filMonomerXFilament(simptr sim,filamentptr fil,char endchar,filamentptr *filptr);`

`void filTreadmill(simptr sim,filamentptr fil,int steps);`

`int filDynamics(simptr sim);`

## 4.12 BioNetGen (functions in smolbng.c)

BioNetGen is a separate software tool that expands biochemical reaction network rules to form complete biochemical reaction networks. Weiren Cui wrote a short perl program, called b2s, which reads BioNetGen output (.net file) and converts that into Smoldyn input. Inspired by it, I wrote a more elaborate interpreter

into the main Smoldyn source code (in C) so that Smoldyn can read these files directly. The plan is that Smoldyn will be distributed with the basic BioNetGen perl program. Then, when Smoldyn encounters rules within an input file, it will automatically call BioNetGen, BioNetGen will expand the rules into a .net file, and Smoldyn will read the .net file into Smoldyn structures.

Data structures declared in smoldyn.h

```
typedef struct bngstruct {
  struct bngsuperstruct *bngss; // bng superstructure
  char *bngname;                // bng name
  int bngindex;                 // index of this bng structure

  int maxparams;                // maximum number of numeric parameters
  int nparams;                  // actual number of numeric parameters
  char **paramnames;            // names of parameters [index]
  char **paramstrings;          // strings for parameter values [index]
  double *paramvalues;          // actual parameter values [index]

  int maxmonomer;               // maximum number of monomers
  int nmonomer;                 // actual number of monomers
  char **monomernames;          // names of monomers [index]
  int *monomercount;            // monomer count work space [index]
  double *monomerdifc;          // diffusion coefficient of monomer [index]
  double *monomerdisplaysize; // display size of monomer [index]
  double **monomercolor;        // color of monomer [index][RGB]
  enum MolecState *monomerstate; // default monomer state [index]
  int bngmaxsurface;            // local copy of nsurface
  enum SrfAction ***monomeraction;  // monomer surface actions [index][srf][face]
  surfactionptr ***monomeractdetails; // monomer action details [index][srf][face]

  int maxbspecies;              // maximum number of bng species
  int nbspecies;                // actual number of bng species
  char **bsplongnames;          // complete bng species names [index]
  char **bspshortnames;         // shortened bng species names [index]
  enum MolecState *bspstate;  // default species state [index]
  char **bspcountstr;           // strings for initial bng species counts [index]
  double *bspcount;             // actual initial bng species counts [index]
  int *spindex;                 // smoldyn index of this species [index]

  int maxbrxns;                 // maximum number of bng reactions
  int nbrxns;                   // acutal number of bng reactions
  char **brxnreactstr;          // strings for reactants [index]
  char **brxnprodstr;           // strings for products [index]
  char **brxnratestr;           // strings for reaction rates [index]
  int **brxnreact;              // reactant bng species indices [index][rct]
  int **brxnprod;               // product bng species indices [index][prd]
  int *brxnorder;               // order of bng reaction [index]
  int *brxnnprod;               // number of products of bng reaction [index]
  rxnptr *brxn;                 // pointer to this reaction [index]
} *bngptr;

typedef struct bngsuperstruct {
  enum StructCond condition;  // structure condition
  struct simstruct *sim;        // simulation structure
  int maxbng;                   // maximum number of bng networks
  int nbng;                     // actual number of bng networks
  char **bngnames;              // names of bng networks
  bngptr *bnglist;              // list of bng networks
} *bngssptr;
```

Superstucture description. As usual, `condition` tells about whether this data structure is up-to-date or not and `sim` points to the simulation structure. `BNG2path` is a string with the path information to the BNG2 program, along with the name of the program (usually BNG2.pl). This superstructure allows for multiple bng networks, the idea being that different macromolecular complexes might be described with separate sets of rules. Each bng network has a name listed in `bngnames` and is listed in `bnglist`. This structure is allocated for `maxbng` possible bng networks, of which `nbng` are currently defined.

Bng structure definition. The bng structure includes a pointer to the bng superstructure that owns it in `bngss`, a pointer to its name in `bngname`, and a copy of its index number in `bngindex`. The `unirate` and `birate` elements are multipliers for unimolecular and bimolecular reaction rates, respectively, which are useful due to the possibility of different units in the BNG file.

To understand the monomer elements, each complex is composed of a collection of monomers, also called seed species (BioNetGen), mols (Moleculizer), or subunits. It is helpful for this parsing software to have a list of these monomers, so they are stored here. The monomers that are listed here are inferred from the species names. There are `nmonomer` monomers, of a total of `maxmonomer` spaces for them. Each monomer has a name. The `monomercount` element is not for storing data, but is purely workspace for the `bngparsespecies` function. Monomer diffusion coefficients, in `monomerdifc` are retrieved from the `sim->mols->difc` data location. The `monomerdisplaysize` and `monomercolor` elements are also retrieved from the `sim->mols` superstructure. The `monomerstate` element stores the default state of the monomer, which is used to compute default states for species and then the states for reactions. The rule is that higher value states take priority over lower value ones (so if a species includes some monomers with default state of `MSsoln` and one of state `MSup`, then the ''up' state will win and that becomes the default state for the species). These monomer lists are sorted so that monomers are listed in alphabetical order. The `bngmaxsurface`, `monomeraction`, and `monomeractdetails` elements are for molecule-surface interactions. The `bngmaxsurface` value is the allocated size of the following two elements, which is made equal to `srfss->nsrf` when things are updated, but might be smaller in the meantime. The next two elements store information about monomer interactions with the surface. The general rule is that more action takes priority over less action.

Most other structure elements correspond to those of the BioNetGen .net file. The species list has allocated size `maxbspecies` and actual size `nbspecies` (the 'b' part of these terms is for BioNetGen, to differentiate these lists from the regular Smoldyn lists). Each species is one that is listed in the .net file. The listed name is put in the `bsplongnames` element and the count from the file is put in the `bspcountstr` element. The name is parsed to monomers and then simplified to a short species name, in `bspshortnames`. The default state is also from the monomers, and is stored in `bspstate`. The species count is parsed into the `bspcount` element. The index of this species within the `sim->mols` data structure is stored in `spindex`.

The reaction list has allocated size `maxbrxns` and actual size `nbrxns`. The reaction data read in from the .net file is separated into reactants in `brxnreactstr`, products in `brxnprodstr`, and the reaction rate in `brxnratestr`. Those items are parsed into bionetgen species numbers (as opposed to Smoldyn species numbers) in the `brxnreact` and `brxnprod` vectors. The reaction has order `brxnorder`, and number of products `brxnnprod`. After the reaction is added to the main Smoldyn program, the pointer to the reaction data structure for this reaction is stored in `brxn`.

Function declarations. As much as possible, the smolbng functions are declared locally rather than in the smoldynfuncs.h header file. This simplifies the code reading because it clarifies which functions might be called externally versus those that are only called internally. Below, all functions are labeled as either "Local" or "Global" to indicate this status.

**Memory management functions**

`void bngallocsurfacedata(bngptr bng,int maxsurface);`

This allocates memory for the surface action elements for the monomers. Enter `bng` with the bng to be updated and `maxsurface` for the desired number of surface spaces (which should be equal to `srfss->nsrf`). This function allocates all of the surface spaces for any monomers that don't have any yet. It also expands the number of surface spaces for monomers that were allocated when there were fewer surfaces. This doesn't return anything except for the updated `bng` structure. If the `bng->bngmaxsurface` element wasn't updated, then this function wasn't able to allocate memory.

`bngptr bngalloc(bngptr bng,int maxparams,int maxbspecies,int maxbrxns);`

Local. Allocates or expands a bng structure, returning a pointer to it. Enter `bng` as `NULL` to allocate a new structure, or as an existing pointer to expand lists within the data structure. Spaces for parameters, species, and reactions are allocated if the entered `max` value is larger than the current one. Returns a pointer to the new or previously existing structure on success, or `NULL` on failure.

`void bngfree(bngptr bng);`
Local. Frees a bng structure, including all of its contents.

`bngssptr bngssalloc(bngssptr bngss,int maxbng);`
Local. Allocates or expands a bngss structure, returning a pointer to it. Enter `bngss` as `NULL` to allocate a new structure, or as an existing pointer to expand lists within the data structure. Spaces for bng names and structures are allocated if the entered `maxbng` value is larger than the current one. Returns a pointer to the new or previously existing structure on success, or `NULL` on failure.

`void bngssfree(bngssptr bngss);`
Local. Frees a bng superstructure, including all of its contents.

**Data structure output**

`void bngoutput(simptr sim);`
Global. Outputs the contents of the bng superstructure and all bng structures to the display.

`int checkbngparams(simptr sim,int *warnptr);`
Global. Checks that the bng parameters are reasonable. This does very little at present.

**Structure set up - bng**

`void bngsetcondition(bngssptr bngss,enum StructCond cond,int upgrade);`
Local. This function sets the condition of the bng superstructure. Set upgrade to 1 if this is an upgrade, to 0 if this is a downgrade, or to 2 to set the condition independent of its current value.

`int bngenablebng(simptr sim,int maxbng);`
Local. This function enables simulation with bng capability. It allocates and initializes the bng superstructure if it doesn't already exist. Send in `maxbng` with -1 for automatic behavior, which is 1 bng structure, or with some other value to specify the number of bng structures. Returns 0 on success or 1 for failure to allocate memory.

`bngptr bngaddbng(simptr sim,const char *bngname);`
Local. Adds a bng structure to the bng superstructure, returning a pointer to it. The new bng structure is named `bngname`. This enables bng function if it hasn't been enabled already. If a bng structure already exists with the same name, then a pointer to that bng is returned.

`int bngsetparam(bngptr bng,char *parameter,double amount);`
Local. Sets the bng structure parameter called `parameter` to the value `amount`. The only options for `parameter` are "unimolecular_rate" and "bimolecular_rate". Returns 0 for success, 1 for illegal `parameter` string, and 2 for illegal `amount` value.

`int bngsetBNG2path(bngptr bng,char *path);`
Local. Sets the bng superstructure `BNG2path` element to the string entered in `path`. Returns 0.

**Structure set up - parameters**

`int bngparseparameter(bngptr bng,int index);`
Local. Parses the parameter value string for the parameter with index `index`, and puts the result into the parameter value element. Returns 0 for success or 1 for failure. If there is a failure, then its description can be found using `strmatherror`.

`int bngaddparameter(bngptr bng,const char *name,const char *string);`
Local. Adds a parameter, or modifies an existing parameter, in a bng structure. Send in the parameter name in `name`. If a parameter of this name doesn't already exist, one is created. The value of `string`, which is allowed to be `NULL` is copied into the `bng->paramstrings` element for parsing to a value. This function sends the new parameter to `bngparseparameter` for parsing. Returns the index of the parameter for success, -1 for failure to allocate memory, or -2 for a math parsing error; in the last case, the error message can be found using `strmatherror`.

**Structure set up - monomers**

`int bngaddmonomer(bngptr bng,const char *name,enum MolecState ms);`
Local. Adds a monomer, or modifies an existing monomer, in a bng structure. Send in the monomer name in `name`. If the default state of this monomer is known, enter it in `ms`; if not, then enter `ms` as `MSsoln`. If a monomer of this name doesn't already exist, one is created. It is added to the list while maintaining alphabetical order. The monomer counts are not changed. Returns the monomer index for success, -1 for failure to allocate memory, or -2 for an invalid monomer name. This sets the monomer diffusion coefficient and color to values in the main Smoldyn code if it is available. In doing so, it looks first for a Smoldyn species name that is the same as the monomer name. If it doesn't find one, it looks for a Smoldyn species name that has two dots and where the portion before the first name is the same as the monomer name (e.g. "X.1.0").

`int bngsetmonomerdifc(bngptr bng,char *name,double difc);`
Local. Sets the diffusion coefficient the monomer called `name` to `difc`. This also adds the monomer to the list if it wasn't there already, and initializes the diffusion coefficient. Returns 0 for success, -1 if out of memory, or -2 for invalid monomer name.

`int bngsetmonomerdisplaysize(bngptr bng,char *name,double displaysize);`
Local. Sets the display size of the monomer called `name` to `displaysize`. This also adds the monomer to the list if it wasn't there already, and initializes the diffusion coefficient. Returns 0 for success, -1 if out of memory, or -2 for invalid monomer name.

`int bngsetmonomercolor(bngptr bng,char *name,double *color);`
Local. Sets the color of the monomer called `name` to `color`, which is a 3-element vector. This also adds the monomer to the list if it wasn't there already, and initializes the diffusion coefficient. Returns 0 for success, -1 if out of memory, or -2 for invalid monomer name.

`int bngsetmonomerstate(bngptr bng,char *name,enum MolecState ms);`
Local. Sets the state of the monomer called `name` to `ms`. This also adds the monomer to the list if it wasn't there already, and initializes the diffusion coefficient. Returns 0 for success, -1 if out of memory, or -2 for invalid monomer name.

**Structure set up - species**

`int bngmakeshortname(bngptr bng,int index,int totalmn,int hasmods);`
Local. Generates a short name for a bspecies, saving it in `bng->bspshortnames`. Send in `index` as the index of this species, `totalmn` as the total number of monomers in this species, and `hasmods` as 1 if the species has at least one modification site and 0 if not. The `monomercount` array needs to be prepared as well. If `totalmn` is 1 and the species has no modification sites, then this species is just a simple unmodifiable monomer; in this case, the short name is the name of the monomer with no suffix. Otherwise, a short name is created which concatenates each monomer name and the number of copies of that monomer, for each monomer, and then adds an isomer code to the end. Always returns 0. This function is only called by `bngparsespecies`. If the short name length as defined here would be longer than the maximum string length, `STRCHAR`, then the length is truncated at a value that is slighly less than this and this species is distinguished from others that have the same name with the isomer number. Thus, errors cannot arise from excessively long strings.

`enum MolecState bngmakedefaultstate(bngptr bng,int index,int totalmn);`
Local. Generates and returns the default state for a bspecies, which has index `index` and total monomers `totalmn`. The `monomercount` array needs to be prepared as well. If the species is also in Smoldyn, then that value is retrieved. If the species has only 1 monomer, the state is the state of that monomer. Otherwise, the state is the greatest state of the monomer states (although with a little re-ordering, so that the priority is `MSsoln`, `MSbsoln`, `MSfront`, `MSback`, `MSup`, `MSdown`. This function is only called by `bngparsespecies`.

`double bngmakedifc(bngptr bng,int index,int totalmn);`
Local. Generates and returns the diffusion coefficient for a bspecies, which has index `index` and total monomers `totalmn`. The `monomercount` array needs to be prepared as well. If the species is also in Smoldyn, then that value is retrieved. If the species has only 1 monomer, then the diffusion coefficient is that of the monomer. Otherwise, it is a weighted average of the monomer diffusion coefficients using the equation $D_{species} = (\sum_i D_i^{-3})^{-1/3}$ where $D_{species}$ is the diffusion coefficient of the species and $D_i$ is the diffusion coefficient of the $i$th monomer within the species. This function is only called by `bngparsespecies`.

`double bngmakedisplaysize(bngptr bng,int index,int totalmn);`
Local. Generates and returns the display size for a bspecies, which has index `index` and total monomers `totalmn`. The `monomercount` array needs to be prepared as well. If the species is also in Smoldyn, then that value is retrieved. If the species has only 1 monomer, then the display size is the display size of that monomer. Otherwise, it is a weighted average of the monomer display sizes using the equation $S_{species} = (\sum_i S_i^3)^{1/3}$ where $S_{species}$ is the display size of the species and $S_i$ is the display size of the $i$th monomer within the species. This function is only called by `bngparsespecies`.

`int bngmakecolor(bngptr bng,int index,int totalmn,double *color);`
Local. Generates and returns the color for a bspecies, which has index `index` and total monomers `totalmn`. The `monomercount` array needs to be prepared as well. If the species is also in Smoldyn, then that value is retrieved. If the species has only 1 monomer, then the color is the color of that monomer. Otherwise, it is a weighted average of the monomer colors. This averaging is weighted by the display sizes of each monomer. This function is only called by `bngparsespecies`.

`void bngmakesurfaction(bngptr bng,int index,int totalmn,enum SrfAction **srfaction,surfactionptr **actdetails);`
Local. Generates and returns the surface actions for a bspecies, which has index `index` and total monomers `totalmn`. The `monomercount` array needs to be prepared as well. If the species is also in Smoldyn, then those values are retrieved. If the species has only 1 monomer, then the surface actions are those of that monomer. Otherwise, it is chosen based on priority, so that greater actions take priority over smaller actions. More precisely, the action ordering is `SAtrans` ¡ `SAmult` ¡ `SAreflect` ¡ `SAjump` ¡ `SAabsorb` ¡ `SAport`. If the only choice is between `SAmult` values, then this goes to the rate values in the action details to decide which has the greater action. This function has barely been tested.

`int bngparsespecies(bngptr bng,int index);`
Local. Parses species with index `index` using its longname. This does lots of things as it goes along. It determines if there are any new monomers, and adds those to the monomer list if so. This also generates a short name for the species, its default state, diffusion coefficient, display size, and color. This species is added to the Smoldyn simulation, including all of its attributes. This then parses the species countstring and adds the correct number of molecules to the Smoldyn simulation. Returns 0 for success, -1 for inability to allocate memory, -2 for a longname that cannot be parsed, -3 for an illegal name (e.g. an asterisk in the name), -4 for a count string that cannot be parsed (in this case, the error can be found from `strmatherror`, or -5 if more molecules are requested than the maximum allowed in Smoldyn, set with `maxdlimit`.

`int bngaddspecies(bngptr bng,int bindex,const char *longname,const char *countstr);`
Local. Adds a species, or modifies an existing species, in a bng structure. The `bindex` value is used to

place the species in the list, meaning that species are not added to the list sequentially, but according to the `bindex` values. This copies `longname` into the `bng->bsplongnames` element and `countstr` into the `bng->bspcountstr` element. Either or both are allowed to be `NULL`. This calls `bngparsespecies` to parse the species, set up all species parameters, and add it to the Smoldyn simulation. Returns 0 for success or the same error codes as `bngparsespecies` for failure.

**Structure set up - reactions**

`int bngparsereaction(bngptr bng,int index);`

Local. Parses the reaction with index `index`. This reads the bng reactant and product names, deals with reactant and product states, and uses this information to create a new Smoldyn reaction. This also reads the reaction rate and uses it to set the Smoldyn reaction rate. Returns 0 for success, 1 for failure to add the reaction to Smoldyn, or 2 for inability to parse the rate string.

`int bngaddreaction(bngptr bng,int bindex,const char *reactants,const char *products,const char *rate);`

Local. Adds a reaction, or modifies an existing reaction, in a bng structure. The `bindex` value is used to place the reaction in the list, meaning that reactions are not added to the list sequentially, but according to the `bindex` values. This copies `reactants` into the `bng->brxnreactstr` element, `products` into the `bng->brxnprodstr` element, and `rate` into the `bng->brxnratestr` element. Any or all are allowed to be `NULL`. This calls `bngparsereaction` to perform all parsing. Returns 0 for success, 1 for failure to allocate memory, or 2 for inability to parse the rate string.

**Structure set up - reactions**

`int bngaddgroup(bngptr bng,int gindex,const char *gname,const char *specieslist);`

Adds a group, created in the .bngl file with "begin observables" and in the .net file as "begin groups" to the Smoldyn simulation, as a species group. Send in the group index as `gindex`. This value is ignored. Send in the group name as `gname` and the species list as `specieslist`. The species list should be a comma-separated list of species indices, using the BioNetGen indices. Returns 0 for success or 1 for errors (the only error that should ever arise is an out of memory error). This function converts the bng species indices to Smoldyn species indices and sends the results to `moladdspeciesgroup` for group creation.

**Structure set up - high level functions**

`int bngrunBNGL2(bngptr bng,char *filename,char *outname);`

Local. This runs the BNG2.pl program on the BNGL file called `filename`. The output file name is returned in `outname`. Returns 0 for success; 1 for inability to find BNG2.pl software at the stored path location; 2 for missing input file called `filename`; or 3 for no output file generated, due to BNG2.pl terminating because of an error in the input file.

`bngptr bngreadstring(simptr sim,ParseFilePtr pfp,bngptr bng,const char *word,char *line2);`

Local. Reads a line of input. This input can start with the word 'name' to give the name of the bng structure. Otherwise, it needs to be a line from a BioNetGen .net file, or a few other words. Returns a pointer to the bng structure for success or `NULL` for failure.

`int loadbng(simptr sim,ParseFilePtr *pfpptr,char* line2);`

Reads BioNetGen .net file, or set of statements. Returns 0 for success or 1 for failure. This function does not require an end statement to stop reading the file, but an end of file serves as well.

`int bngupdateparams(simptr sim);`

Local. Doesn't do anything at the moment.

`int bngupdatelists(simptr sim);`

Local. This does nothing, simply returning 0 to indicate success.

`int bngupdate(simptr sim);`

Updates a bng structure, bringing it up to the ok condition. This calls the `bngupdatelists` and `bngupdateparams` functions to carry out the updating tasks.

**Core simulation functions**

**No core simulation functions.**

## 4.13 Complexes (not written yet)

It's becoming increasingly apparent that Smoldyn needs to support macromolecular complexes. This section presents documentation for code that hasn't been written yet in the hopes that this will provide a design for the code once there is time to write it.

Data structures

Each individual complex is listed with a complexstruct, and this complexstruct lists each of its monomers individually in a monomerstruct.

```
typedef struct monomerstruct {
  struct complexstruct *cmplx;      // owning complex superstructure
  moleculeptr *mptr;          // molecule that is this monomer
  double *dispsph;          // displacement of monomer in r,q,f,x
  double *dispcart;         // displacement of this monomer in x,y,z
  struct monomerstruct **site;      // monomer at each binding site [bs]
  int *shape;           // shape of each binding site [bs]
  } *monomerptr;

typedef struct complexstruct {
  struct complexsuperstruct *cmplxss;   // owning complex superstructure
  int maxmonomer;         // maximum number of monomers
  int nmonomer;           // actual number of monomers
  monomerptr *monomers;       // list of component monomers
  double *pos;            // center of mass position [d]
  double *posx;           // old COM position [d]
  double *rotation;         // complex rotation in q,f,x
  double mass;            // complex mass
  double difc;            // complex diffusion coefficient
  double difstep;           // complex rms step length
  } *complexptr;

typedef struct complexsuperstruct{
  struct simstruct *sim;          // owning simulation structure
  int maxcomplex;         // maximum number of complexes
  int ncomplex;           // actual number of complexes
  complexptr *complexes;        // list of individual complexes
// rules about connectivity
  int *nsites;              // number of sites on species [i]
  int **nshapes;            // number of shapes [i][bs]
  double ***shapemass;        // mass of shape [i][bs][sh]
  char ****shapename;       // shape names [i][bs][sh]
  double ****dispsph;         // site displacement in r,q,f,x, [i][bs][sh]
// rules for complex names (i.e. explicit-species and species-classes)

  } *complexssptr;
```

Add to moleculesuperstruct: double *mass; // mass of species [i]

## 4.14 Graphics (functions in smolgraphics.c)

Overall, Smoldyn's graphics use is fairly straightforward, although it is nevertheless a little complicated due to the design of the OpenGL glut library. For stand-alone Smoldyn, the program's entry and exit point are in the `main` function, which is in smoldyn.c. From here, the program forks to either `smolsimulate` (in smolsim.c) if graphics are not used or to `smolsimulategl` (in smolgraphics.c) if graphics are used. In the latter case, the OpenGL glut framework is used, in which control is passed from the main program to OpenGL and is not returned again until the user chooses quit from a menu. This leads to a slightly strange program structure, although I have attempted to contain all of the strangeness to a few functions at the end of smolgraphics.c.

If graphics are used, then `smolsimulategl` sets up a few graphics options that will apply for the entire simulation, calls `gl2Initialize` with the system boundaries for more overall set up, sets the background color, registers `RenderScene` as the display callback function, registers `TimerFunction` as the simulation callback function, and then goes into the black box called `glutMainLoop`. The only graphics done in `TimerFunction` is that it posts a need for graphical update on occasion with `glutPostRedisplay`. Meanwhile, `RenderScene` is just a wrapper for `RenderSim`, which actually draws the entire graphical output. The summary is: initialization is done in `smolsimulategl` and drawing is done by `RenderSim`.

Data structure

```
#define MAXLIGHTS 8;
enum LightParam {LPambient,LPdiffuse,LPspecular,LPposition,LPon,LPoff,LPauto,
    LPnone};

typedef struct graphicssuperstruct {
  int graphics;               // graphics: 0=none, 1=opengl, 2=good opengl
  int currentit;              // current number of simulation time steps
  int graphicit;              // number of time steps per graphics update
  unsigned int graphicdelay;    // minimum delay (in ms) for graphics updates
  int tiffit;                 // number of time steps per tiff save
  double framepts;            // thickness of frame for graphics
  double gridpts;             // thickness of virtual box grid for graphics
  double framecolor[4];       // frame color [c]
  double gridcolor[4];        // grid color [c]
  double backcolor[4];        // background color [c]
  double textcolor[4];        // text color [c]
  int maxtextitems;           // allocated size of item list
  int ntextitems;             // actual size of item list
  char **textitems;           // items to display with text [item]
  double ambient[4];          // global ambient light [c]
  int lightstate[MAXLIGHTS];    // whether light is on or off [lt]
  double ambilight[MAXLIGHTS][4];  // ambient light color [lt][c]
  double difflight[MAXLIGHTS][4];  // diffuse light color [lt][c]
  double speclight[MAXLIGHTS][4];  // spectral light color [lt][c]
  double lightpos[MAXLIGHTS][3];   // light positions [lt][d]
  } *graphicssptr;
```

**enumerated types**

`enum LightParam graphicsstring2lp(char *string);`
Converts a string to an enumerated light parameter.

`char *graphicslp2string(enum LightParam lp,char *string)`
Converts an enumerated light parameter to a string. The string is returned.

**low level utilities**

`int graphicsreadcolor(char **stringptr,double *rgba);`
Reads the text of the string that `stringptr` points to, to find color information. The data are returned

in the vector `rgba`, if `rgba` is not sent in as `NULL`. The input data are in a pointer to a string, rather than just a string, so that the string can be advanced to the end of the color information. Upon return, if this function is successful, the contents of `stringptr` points to the first word of the string that follows the color information, or to `NULL` if the end of the string was reached while parsing color information. If `rgba` is supplied, it needs to be allocated to hold at least 4 numbers, which are for the red, green, blue, and alpha color channels, respectively. The string needs to list the color either with three space-separated numbers, each between 0 and 1 inclusive, or with a single word. Word options are: maroon, red, orange, yellow, olive, green, purple, magenta, lime, teal, cyan, blue, navy, black, gray, silver, and white. Other words are not recognized. Following the color information, the string can optionally list the alpha value, as a number between 0 and 1. If alpha is not listed, a default value of 1 is assigned. The function returns 0 for no error, 1 if `string` is missing or is empty, 2 if too few numbers are listed, 3 if one or more of the listed color numbers is out of range (the listed numbers are returned in `rgba`), 4 if a word was given but it isn't recognized, 5 if an alpha value was given but can't be parsed, or 6 if the listed alpha value is out of range.

**memory management**

`graphicsssptr graphssalloc(void)`
Allocates and intializes the graphics superstructure. No OpenGL stuff is intialized here.

`void graphssfree(graphicsssptr graphss)`
Frees a graphics superstructure.

**data structure output**

`void graphssoutput(simptr sim)`
Displays all graphics parameters from the graphics superstructure to stdout. Also displays some information from the opengl2 library variables, including the TIFF name and TIFF numbering.

`void writegraphss(simptr sim,FILE *fptr)`
Writes graphics information to `fptr` as part of a Smoldyn-readable input file.

`int checkgraphicsparams(simptr sim,int *warnptr)`
Checks graphics parameters for actual or possible errors. Returns the number of errors directly and returns the number of warnings in `warnptr`, if `warnptr` isn't `NULL`. At present, this doesn't check anything, but just returns two zeros.

**structure setup**

`void graphicssetcondition(graphicsssptr graphss,enum StructCond cond,int upgrade);`
Sets the graphics superstructure condition to `cond`, if appropriate. Set `upgrade` to 1 if this is an upgrade, to 0 if this is a downgrade, or to 2 to set the condition independent of its current value. If the condition is downgraded, this also downgrades the simulation structure condition.

`int graphicsenablegraphics(simptr sim,char *type);`
Enables graphics by allocating a graphics superstructure and adding it to the simulation structure. Enter `type` as "none" for no graphics, "opengl" for minimal OpenGL graphics (molecules are square dots), "opengl_good" for reasonably good OpenGL graphics (molecules are solid colored spheres), "opengl_better" for better OpenGL graphics (use of lighting and shininess), or `NULL` for default enabling which is no change if graphics already exist and basic OpenGL if they don't already exist. Returns 0 for success, 1 for inability to allocate memory, 2 for missing `sim` input, or 3 for an invalid `type` string.

`int graphicssetiter(simptr sim,int iter);`
Sets the `graphicit` element of the graphics superstructure, which tells how many simulation iterations should be allowed to pass between graphics renderings. This enables graphics, if needed. Returns 0 for success, 1 for out of memory enabling graphics, 2 for no `sim`, or 3 for an illegal `iter` value (it needs to be at least 1).

`int graphicssetdelay(simptr sim,int delay);`
Sets the `graphicdelay` element of the graphics superstructure, which gives the minimum number of milliseconds that should be allowed to elapse between graphics renderings, to keep simulations from running too fast to see. This enables graphics, if needed. Returns 0 for success, 1 for out of memory enabling graphics, 2 for no `sim`, or 3 for an illegal `delay` value (it needs to be at least 0).

`int graphicssetframethickness(simptr sim,double thickness);`
Sets the `framepts` element of the graphics superstructure, which gives the drawing thickness of the frame that surrounds the simulation volume. This enables graphics, if needed. Returns 0 for success, 1 for out of memory enabling graphics, 2 for no `sim`, or 3 for an illegal `thickness` value (it needs to be at least 0).

`int graphicssetframecolor(simptr sim,double *color);`
Sets the `framecolor` elements of the graphics superstructure, which gives the color of the frame that surrounds the simulation volume. Color is a four-element vector (red, green, blue, alpha). This enables graphics, if needed. Returns 0 for success, 1 for out of memory enabling graphics, 2 for no `sim`, or 3 for one or more illegal `color` values (they need to be between 0 and 1, inclusive).

`int graphicssetgridthickness(simptr sim,double thickness);`
Sets the `gridpts` element of the graphics superstructure, which gives the drawing thickness of the partitions that separate the virtual boxes. This enables graphics, if needed. Returns 0 for success, 1 for out of memory enabling graphics, 2 for no `sim`, or 3 for an illegal `thickness` value (it needs to be at least 0).

`int graphicssetgridcolor(simptr sim,double *color);`
Sets the `gridcolor` elements of the graphics superstructure, which gives the color of the partitions that separate the virtual boxes. Color is a four-element vector (red, green, blue, alpha). This enables graphics, if needed. Returns 0 for success, 1 for out of memory enabling graphics, 2 for no `sim`, or 3 for one or more illegal `color` values (they need to be between 0 and 1, inclusive).

`int graphicssetbackcolor(simptr sim,double *color);`
Sets the `backcolor` elements of the graphics superstructure, which gives the color of the background. Color is a four-element vector (red, green, blue, alpha). This enables graphics, if needed. Returns 0 for success, 1 for out of memory enabling graphics, 2 for no `sim`, or 3 for one or more illegal `color` values (they need to be between 0 and 1, inclusive).

`int graphicssettextcolor(simptr sim,double *color);`
Sets the `textcolor` elements of the graphics superstructure, which gives the color of text drawn to the graphics window. Color is a four-element vector (red, green, blue, alpha). This enables graphics, if needed. Returns 0 for success, 1 for out of memory enabling graphics, 2 for no `sim`, or 3 for one or more illegal `color` values (they need to be between 0 and 1, inclusive).

`int graphicssettextitem(simptr sim,char *itemname);`
Adds an item to the list of things that Smoldyn will display to the graphics window. Enter the name of the item as a string in `itemname`. This automatically allocates space for text items as needed. Returns 0 for success, 1 if memory could not be allocated, 2 if the item is not a supported string, or 3 if the item was already listed. Currently supported names are: "time" and species(state) names.

`int graphicssetlight(simptr sim,graphicsssptr graphss,int lt,enum LightParam ltparam,double *value)`
Sets parameters for the lighting portions of the graphics superstructure. If `graphss` is entered as non-`NULL`, then it is worked with and `sim` is ignored; otherwise, this requires the `sim` input and enables graphics if needed. `lt` is the light number, which should be between 0 and 8, or set `lt` to -1 for the global ambient light source. `ltparam` is the lighting parameter that should be set. `It` is only allowed to be `LPambient` if lt is -1. Otherwise, `ltparam` can be `LPambient`, `LPdiffuse`, `LPspecular` and then value should list the 4 color values. Or, `ltparam` can be `LPposition` and `value` should list the three light position values. Or, `ltparam` can be `LPon` or `LPoff` to turn the light on or off, in which case `value`

is ignored. No checking is done to see that input parameters are legitimate. Returns 0 for success or 1 if memory could not be allocated for the graphics superstructure.

**structure update functions**

`int graphicsupdateinit(simptr sim);`
Performs basic graphics initialization. This calls `gl2glutInit` to initialize things and then `gl2Initialize` to create the graphics window and set the viewing coordinates. This function should probably be called only once.

`int graphicsupdatelists(simptr sim);`
Enables lighting models for the graphics display. This only needs to be called when the user requests the "opengl_better" graphics option.

`int graphicsupdateparams(simptr sim);`
Updates the lighting model parameters for the graphics display. This should be called every time the user changes the lighting model. It also sets the background color.

`int graphicsupdate(simptr sim);`
Updates the graphics superstructure and upgrades the graphics condition element. This calls `graphicsupdateinit`, `graphicsupdatelists`, and/or `graphicsupdateparams`, depending on the amount of updating required.

**core simulation functions**

`void RenderSurfaces(simptr sim)`
Draws all surfaces in the simulation using OpenGL graphics. The 3-D portion of this function needs some work, both to fix 3-D disk drawing, to improve 3-D drawing overall, and for overall cleanup.

`void RenderMolecs(simptr sim)`
Draws all molecules using OpenGL graphics. Because the molecules are not sorted by type, this function is fairly inefficient; this could be a significant computational burden for movie-making, but shouldn't be for most research purposes.

`void RenderText(simptr sim);`
Draws any requested text to the OpenGL graphics window.

`void RenderSim(simptr sim)`
Draws the entire graphical output using OpenGL graphics. This calls other functions for most of the work, although it draws the frame and the grid itself.

**Top level OpenGL functions** Both `RenderScene` and `TimerFunction` are declared locally, rather than in smoldynfuncs.h. This makes them invisible outside of this source file. They are callback functions for OpenGL. In addition, the `Sim` variable is declared as a global variable, with the scope of this file. It is here because OpenGL does not allow `void*` pointers to be passed through to all callback functions, so making it a global variable enables the callback functions to access the simulation data structure.

`void RenderScene(void);`
`RenderScene` is the call-back function for OpenGL that displays the graphics. This does nothing but call `RenderSim`.

`void TimerFunction(int state);`
`TimerFunction` is the call-back function for OpenGL that runs the simulation. `state` is positive if the simulation should quit due to a simulation error or normal ending, `state` is negative if the simulation has been over, and `state` is 0 if the simulation is proceeding normally. This also looks at the state defined in the opengl2 library; if it is 0, the simulation is continuing, if it is 1, the simulation is in pause mode, and if it is 2, the user told the simulation to quit. `oldstate` is the old version of the `gl2State` value. This function runs one simulation time step, posts graphics redisplay flags, and saves TIFF files as appropriate.

| oldstate | state | gl2State | meaning | next state |
|---|---|---|---|---|
| 1 | - | 0 | leave pause state | (0 - 0) |
| 0 | 0 | 0 | continue simulating | (0 =simstep 0) |
| - | >0 | - | stop the simulation | (- -1 -) |
| - | 0 | 2 | " " " | (- -1 2) |
| 0 | 0/-1 | 1 | enter pause state | (1 - 1) |
| - | 0/-1 | - | in pause state or sim is over | (- - -) |

`void smolsimulategl(simptr sim);`

`smolsimulategl` initiates the simulation using OpenGL graphics. It does all OpenGL initializations, registers OpenGL call-back functions, sets the global variables to their proper values, and then hands control over to OpenGL. This function returns as the program quits.

## 4.15 Simulation structure (functions in smolsim.c)

At the highest level of the structures is the simulation structure. This is a large framework that contains information about the simulation that is to be run as well as pointers to each of the component structures and superstructures. It also contains some scratch space for functions to use as they wish.

### Data structures

```
#define ETMAX 10
enum SmolStruct {SSmolec,SSwall,SSrxn,SSsurf,SSbox,SScmpt,SSport,SScmd,SSmzr,SSsim
    ,SScheck,SSall,SSnone};
enum EventType {ETwall,ETsurf,ETdesorb,ETrxn0,ETrxn1,ETrxn2intra,ETrxn2inter,
    ETrxn2wrap,ETimport,ETexport};

typedef void (*logfnptr)(struct simstruct *,int,const char*,...);
typedef int (*diffusefnptr)(struct simstruct *);
typedef int (*surfaceboundfnptr)(struct simstruct *,int);
typedef int (*surfacecollisionsfnptr)(struct simstruct *,int,int);
typedef int (*assignmols2boxesfnptr)(struct simstruct *,int,int);
typedef int (*zeroreactfnptr)(struct simstruct *);
typedef int (*unimolreactfnptr)(struct simstruct *);
typedef int (*bimolreactfnptr)(struct simstruct *,int);
typedef int (*checkwallsfnptr)(struct simstruct *,int,int,boxptr);

typedef struct simstruct {
  enum StructCond condition;  // structure condition
  logfnptr logfn;             // function for logging output
  FILE *logfile;              // file to send output
  char *filepath;             // configuration file path
  char *filename;             // configuration file name
  char *flags;                // command-line options from user
  time_t clockstt;            // clock starting time of simulation
  double elapsedtime;         // elapsed time of simulation
  long int randseed;          // random number generator seed
  int eventcount[ETMAX];      // counter for simulation events
  int dim;                    // dimensionality of space.
  double accur;               // accuracy, on scale from 0 to 10
  double time;                // current time in simulation
  double tmin;                // simulation start time
  double tmax;                // simulation end time
  double dt;                  // simulation time step
  int quitatend;          // 1 if quit simulation at end
  rxnssptr rxnss[MAXORDER];   // reaction superstructures
```

```
molssptr mols;              // molecule superstructure
wallptr *wlist;             // list of walls
surfacessptr srfss;         // surface superstructure
boxssptr boxs;              // box superstructure
compartssptr cmptss;        // compartment superstructure
portssptr portss;           // port superstructure
mzrssptr mzrss;             // network generation rule superstructure
void* cmds;             // command superstructure
graphicssptr graphss;       // graphics superstructure
threadssptr threads;        // pthreads superstructure
diffusefnptr diffusefn;                     // function for molecule diffusion
surfaceboundfnptr surfaceboundfn;           // function for surface-bound
    molecules
surfacecollisionsfnptr surfacecollisionsfn; // function for surface collisons
assignmols2boxesfnptr assignmols2boxesfn;   // function that assigns molecs to
    boxes
zeroreactfnptr zeroreactfn;                 // function for zero order reactions
unimolreactfnptr unimolreactfn;             // function for first order
    reactions
bimolreactfnptr bimolreactfn;               // function for second order
    reactions
checkwallsfnptr checkwallsfn;               // function for molecule collisions
    with walls
} *simptr;
```

`ETMAX` is the maximum number of event types, which are enumerated with `EventType`. These are used primarily for reporting the number of times that various things happened to the user, although they are also used occasionally elsewhere in the code so that certain routines are only done if they are necessary. `SmolStruct` enumerates the different types of superstructures that a simulation can have. It does not appear to be used anywhere in the code, so I'm not sure why I created it.

The list of function pointers defined with typedef statements are used below in the `simstruct`. They make it possible for a simulation to use different collections of core algorithms. The first one, logfnptr, is for an external logging function, for use by Libsmoldyn. Others allow the use of either single-threaded or multi-threaded algorithm versions. Also, which has not been done but could be, they could be tuned for better efficiency in, say, a 3D system, rather than being generalists for all system dimensionalities.

`simstruct` contains and owns all information that defines the simulation conditions, the current state of the simulation, and all other simulation parameters. The `condition` element is maintained at the lowest level of all superstructure conditions. The `logfn` is typically set to `NULL` but can be set to point to a function if text output should be dealt with there instead of in simLog. `logfile` is used by simLog for sending all output. The default is stdout. The `filepath` and `filename` give the configuration file name and `flags` lists the command-line flags that the user supplied. `clockstt` is used for the clock value when the simulation starts, and `elapsedtime` is used for storing the simulation run time while the simulation is paused, both of which are for timing simulations. `randseed` is the starting random number seed. `eventcount` is a list of counts for each of the enumerated event types. `dim` is the system dimensionality and `accur` is the overall simulation accuracy level. Because this has not proven useful, it should be removed at some point, and a version of it should be moved to the box superstructure. Finally, `time`, `tmin`, `tmax`, and `dt` are the current time, starting time, stopping time, and time step of the simulation, respectively. `tbreak` is the simulation break time, which lets functions that use Libsmoldyn run the simulation for a fixed amount of time and then stop for other operations. `quitatend` is a simple flag that is 1 if the simulation should simply quit when done and 0 if not. This is only relevant for simulations with graphics, because the others automatically quit at the end anyhow.

The superstructures are listed next. All of them are optional, where a `NULL` value simply means that the simulation does not include that feature and an existing superstructure means that the simulation has that feature. The command superstructure is pointed to with a `void*` rather than a `cmdssptr` because the latter is declared in a separate header file and I didn't want to require a dependency between the smoldyn.h header file and the SimCommand.h header file.

Finally, the simulation structure lists the function pointers for the core simulation algorithms.

## Functions

### enumerated types

`enum SmolStruct simstring2ss(char *string)`
Returns the enumerated simulation structure type that corresponds to the string input. Returns `SSnone` if input is "none" or if it is not recognized.

`char *simss2string(enum SmolStruct ss,char *string)`
Returns the string that corresponds to the enumerated simulation structure input in `string`, which needs to be pre-allocated. The address of the string is returned to allow for function nesting.

`char *simsc2string(enum StructCond sc,char *string)`
Returns the string that corresponds to the enumerated structure condition input in `string`, which needs to be pre-allocated. The address of the string is returned to allow for function nesting.

### low level utilities

`double simversionnumber(void);`
Returns the version number of Smoldyn. This reads the `VERSION` string from smoldyn_config.h into a double and returns that value. A value of 0 indicates that the reading didn't work.

`void Simsetrandseed(simptr sim,long int randseed)`
Sets the random number generator seed to `seed` if `seed` is at least 0, and sets it to the current time value if `seed` is less than 0.

### memory management

`simptr simalloc(char *root)`
Allocates a simulation structure. Essentially everything, including superstructures, is initialized to 0 or `NULL`. Exceptions are that the `filepath`, `filename`, and `flags` strings are allocated, the random number seed is initialized with a random value, and the command superstructure is allocated. `root` is a required input because it is sent to the command superstructure allocation; it is allowed to be `NULL`.

`void simfree(simptr sim)`
`simfree` frees a simulation strucutre, including every part of everything in it.

`void simfuncfree(void);`
Frees memory that is allocated by functions within the Smoldyn program and that is only kept track of by the functions themselves, using static variables. This should be called just before program termination.

`int simexpandvariables(simptr sim,int spaces);`
Expands the number of variables in a simulation structure by `spaces` spaces. This allocates memory, copies over existing variables, and clears new ones as needed.

### data structure output

`void simLog(simptr sim,int importance,const char* format, ...)`
All text output should be sent to this function. As a default, it simply prints the output to stdout. However, this also sends it elsewhere if the user asked for a different destination. This can also send output to a Libsmoldyn host program. Enter `sim` with the simulation structure if possible; if it's not possible, then this function displays the message to stderr. Enter `importance` with a value between 0 and 10, as shown below. The `format` and `...` portions are the same format string and arguments that are used for `printf`.

| `importance` | meaning | example | flags and display |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 0 | debugging output | | v enables |
| 1 | verbose output | box details | v enables |
| 2 | normal diagnostics | reaction parameters | q suppresses |
| 3 | abbreviated diagnostics | basic species | q suppresses |
| 4 | important notices | simulation has ended | q suppresses |
| 5 | warnings | time step too big | w suppresses |
| 7 | minor errors | command errors<br>structure condition not updated | w suppresses |
| 8 | recoverable errors | user input syntax error | always shown |
| 9 | error in normal operation | child structures missing parents | always shown |
| 10 | unrecoverable errors | bug or memory allocation failure | always shown |

`void simoutput(simptr sim)`
: `simoutput` prints out the overall simulation parameters, including simulation time information, graphics information, the number of dimensions, what the molecule types are, the output files, and the accuracy.

`void simsystemoutput(simptr sim);`
: Displays information about all components of the simulation to stdout by calling output functions for each superstructure.

`void writesim(simptr sim,FILE *fptr)`
: Writes all information about the simulation structure, but not its substructures, to the file `fptr` using a format that can be read by Smoldyn. This allows a simulation state to be saved.

`int checksimparams(simptr sim)`
: `checksimparams` checks that the simulation parameters, including parameters of sub-structures, have reasonable values. If values seem to be too small or too large, a warning is displayed to the standard output, although this does not affect continuation of the program. Returns the number of errors.

**structure set up**

Initialization procedures are meant to be called once at the beginning of the program to allocate and set up the necessary structures. These routines call memory allocation procedures as needed. `simupdate` is the only one of these routines that should ever need to be called externally, since it calls the other functions as needed.

`int simsetpthreads(simptr sim,int number);`
: Sets the number of pthreads that the simulation should run in to `number`. Send in `number` as 0 for unthreaded mode, which is the default, or as a larger value for threaded operation. A value of 1 will causes multi-threaded operation, but with 1 thread. If threading is requested and the function is able to fulfill it, it returns the number of threads. If threading is not requested, the function returns 0. If threading is requested and the function fails because it's not enabled, then it returns -1, and if it fails because of failure to allocate memory, it returns -2.

`void simsetcondition(simptr sim,enum StructCond cond,int upgrade);`
: Sets the simulation structure condition to `cond`, if appropriate. Set `upgrade` to 1 if this is an upgrade, to 0 if this is a downgrade, or to 2 to set the condition independent of its current value.

`int simsetvariable(simptr sim,char *name,double value);`
: Sets the value of the variable named `name` to `value`. This creates the variable if it doesn't already exist. This also allocates memory for new variables as neeed. Returns 0 for success and 1 for out of memory.

`int simsetdim(simptr sim,int dim)`
Sets the simulation dimensionality. Returns 0 for success, 2 if it had already been set (it's only allowed to be set once), or 3 if the requested dimensionality is not between 1 and 3.

`int simsettime(simptr sim,double time,int code)`
Sets the appropriate simulation time parameter to `time`. Enter code as 0 to set the current time, 1 to set the starting time, 2 to set the stopping time, 3 to set the time step, or 4 to set the break time. Returns 0 for success, 1 if an illegal code was entered, or 2 if a negative or zero time step was entered. This function also keeps track of the times that have been set using a static variable called `timedefined`. To see what times have been set, enter code as -1, and this will return a number which is the sum of: 1 for the current time, 2 for the starting time, 4 for the stopping, 8 for the time step, and 16 for the break time. For example, and the return value with 14 to check for the start, stop, and step times.

`int simreadstring(simptr sim,char *word,char *line2,char *erstr)`
Reads and processes one line of text from the configuration file, or some other source. The first word of the line should be sent in as `word` (terminated by a '\0') and the rest sent in as `line2`. I don't think that this function changes either the contents of `word` or `line2`, but this should be verified if it's important. If this function is successful, it returns 0 and it does not change the contents of `erstr`; if not, it returns 1 and it writes an error message to `erstr`.

`int loadsim(simptr sim,char *fileroot,char *filename,char *erstr,char *flags)`
`loadsim` loads all simulation parameters from a configuration file, using a format described above. `fileroot` is sent in as the root of the filename, including all colons, slashes, or backslashes; if the configuration file is in the same directory as Smoldyn, `fileroot` should be an empty string. `filename` is sent in as just the file name and any extension. `erstr` is sent in as an empty string of size `STRCHAR` and is returned with an error message if an error occurs. `sim` is the simulation structure. This routine calls `loadsurface` to load any surfaces. The following things are set up after this routine is completed: all molecule elements except box; all molecule superstructure elements; all wall elements; box superstructure element `mpbox`, but no other elements; no boxes are allocated or set up; all reaction structure elements except `rate2` and the product template position vectors (`pos` in each product); the command superstructure, including all of its elements; and all simulation structure elements except for sub-elements that have already been listed. All new molecules are left in the dead list for sorting later. If the configuration file loads successfully, the routine returns 0. If the file could not be found, it returns 10 and an error message. If an error was caught during file loading, the return value is 10 plus the line number of the file with an error, along with an error message. If there is an error, all structures are freed automatically.

`int simupdate(simptr sim,char *erstr)`
Updates all parts of the simulation structure. This is called on start up by `setupsim`, and may be called at anytime afterwards. It returns 0 for success or 1 for failure. In the latter case, a string that describes the error should be returned in `erstr`.

`int setupsim(char *root,char *name,simptr *smptr,char *flags)`
`setupsim` sets up and loads values for all the structures as well as global variables. This routine calls the other initialization routines, so they do not have to be called from elsewhere. It also displays the status to stdout and calls output routines for each structure, allowing verification of the initiallization. Normally, send in `root` and `name` with strings for the path and name of the input file and send in `smptr` (pointer to a simulation structure) pointing to a `NULL`. `flags` is a string of command-line flags. This returns 0 for correct operation and 1 for an error. If it succeeds, `smptr` is returned pointing to a fully set up simulation structure. Otherwise, `smptr` is set to `NULL` and an error messages is displayed on stderr. In the alternate use, send in `root` and `name` as `NULL` and send in `smptr` pointing to a partially set up simulation structure; it should be set up to the same extent that it is after it is returned from `loadsim`. With this alternate input, this function will finish setting up the simulation structure.

**core simulation functions**

`int simdocommands(simptr sim);`
Performs all commands that should happen at the current time. This includes commands that should happen before or after the simulation. This function leaves data structures in good shape. Returns 0 to indicate that the simulation should continue, 6 for error with `molsort`, 7 for terminate instruction from `docommand`, or 8 for failed simulation update. These are the same error codes that `simulatetimestep` uses.

`int simulatetimestep(simptr sim)`
`simulatetimestep` runs the simulation over one time step. If an error is encountered at any step, or a command tells the simulation to stop, or the simulation time becomes greater than or equal to the requested maximum time, the function returns an error code to indicate that the simulation should stop; otherwise it returns 0 to indicate that the simulation should continue. Error codes are 1 for simulation completed normally, 2 for error with `assignmolecs`, 3 for error with `zeroreact`, 4 for error with `unireact`, 5 for error with `bireact`, 6 for error with `molsort`, 7 for terminate instruction from `docommand` (e.g. stop command), 8 for failed simulation update, 9 for error with `diffuse`, 10 for simulation stopped because the time equals or exceeds the break time, 11 for error in filament dynamics, 12 for error in lattice simulation, or 13 for error in reaction network expansion. Errors 2 and 6 arise from insufficient memory when boxes were being expanded and errors 3, 4, and 5 arise from too few molecules being allocated initially.

Note that the sequence in which the simulation components are set up is designed carefully and will likely lead to bugs if it is changed.

`void endsimulate(simptr sim,int er)`
`endsimulate` takes care of things that should happen when the simulation is complete. This includes executing any commands that are supposed to happen after the simulation, displaying numbers of simulation events that occurred, and calculating the execution time. `er` is a code to tell why the simulation is ending, which has the same values as those returned by `simulatetimestep`. If graphics are used, this routine just returns to where it was called from (which is `TimerFunction`); otherwise, it frees the simulation structure and then returns (to `smolsimulate` and then `main`).

`int smolsimulate(simptr sim);`
`smolsimulate` runs the simulation without graphics. It does essentially nothing other than running `simulatetimestep` until the simulation terminates or stops due to reaching the break time.

## 4.16 Commands (functions in smolcmd.c)

### Writing commands

Command strings are not parsed, checked, or even looked at during simulation initialization. Instead, they are run by the command interpreter during the simulation. Command routines are given complete freedom to look at and/or modify any part of a simulation structure or sub-structure. This, of course, also gives commands the ability to crash the computer program, so they need to be written carefully to prevent this. Every command is sent a pointer to the simulation structure in sim, as well as a string of command parameters in `line2`.

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

- Write a description of the new command that will go into the reference section of the users manual.
- 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);
```

- 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);
```

- Write the function for the new command, modeling it on the command functions currently in smolcmd.c. See below.
- Proofread the function and test the command.
- Write documentation about the command for this section of this manual.
- Mention the command in the modifications portion of this manual.

Each command is written with a similar structure. As an example, here is `cmdecho`:

```
enum CMDcode cmdecho(simptr sim,cmdptr cmd,char *line2) {
  FILE *fptr;
  char *termqt,str[STRCHAR];

  if(line2 && !strcmp(line2,"cmdtype")) return CMDobserve;
  fptr=scmdgetfptr(sim->cmds,line2);
  SCMDCHECK(fptr,"file␣name␣not␣recognized");
  line2=strnword(line2,2);
  SCMDCHECK(line2=strchr(line2,'"'),"no␣starting␣quote␣on␣string");
  strncpy(str,line2+1,STRCHAR-1);
  str[STRCHAR-1]='\0';
  SCMDCHECK(termqt=strchr(str,'"'),"no␣terminal␣quote␣on␣string");
  *termqt='\0';
  strbslash2escseq(str);
  scmdfprintf(cmd->cmds,fptr,"%s",str);
  scmdflush(fptr);
  return CMDok; }
```

Every command has essentially the same function call, where the first parameter is the simulation structure, the second parameter is the command structure of the current command, and the third parameter is the remainder of the input line from this command entry in the user's configuration file.

The first line of code checks to see whether `line2` is entered as "cmdtype" (which is never a valid entry from the user) and if so, returns the type of the current command. These types are `CMDcontrol` for commands that control the running of the simulation, `CMDobserve` for commands for observing the system but that do not change it, and `CMDmanipulate` for commands that modify the simulation state but do not have any output. Conditional commands, all of which have an "if" in the name, return the command type of the command that they call if the condition is met (see `cmdifflag`).

The body of the command is after this first line of code. In the body, the command parses `line2` while checking for valid input, does the requested action, flushes any output file (which is opened and closed elsewhere), and then returns `CMDok` to indicate that it terminated successfully. Along the way, any errors are trapped with the `SCMDCHECK` macro. To isolate commands from the user interface, they get file pointers by calling `scmdgetfptr`, they write to the file using `scmdfprintf`, and they flush the file with `scmdflush`.

Commands that output data to file should use the `scmdfprintf` function. This function handles output precision automatically. Also, you should separate data values using the `%,` formatting symbol, which the `scmdfprintf` function converts to either a space or a comma, depending on whether the user wants space-separated vectors or comma-separated vectors.

Commands that read numbers from user input, whether integers or floating point values should not do so with `sscanf` but should use `strmathsscanf` instead. This is a simple replacement for `sscanf` but it evaluates any formulas that the user provides for numerical input. To specify that formala evaluation should be enabled for a specific numerical input, replace the %i format symbol with %mi and replace %lg with %mlg. The function call also requires the simulation variable list. These are available as global variables, as `Varnames`, `Varvalues`, and `Nvar`.

Commands that read molecule species should do so using the `string2index1` function. This reads a string for a species name and an optional state, and then returns the species index and the state. Also, if the user did not ask for a single species but for a group of species, then this returns the full list of species in this group. It also returns error codes. The variety of outputs would normally be somewhat annoying, which is what `molscan` was written to handle, described next.

A lot of commands scan over all molecules, whether to count the molecules, do other statistics on them, or other things. This looping is already somewhat complicated, and much more so if commands allow the user to enter species names with wildcards or species group names. Furthermore, commands are supposed to support these inputs if at all possible. To simplify the code in the commands, the molecule iteration loop has been written in `molscan`, and that function then calls back to the command to actually do whatever needs to be done with the identified molecules. This leads to a more complicated command structure, of which an example is in `cmdifincmpt`, which calls another command if there are some number of molecules in a specified compartment. Here is part of its listing:

```
enum CMDcode cmdifincmpt(simptr sim,cmdptr cmd,char *line2) {
  ... variable declarations ...
  moleculeptr mptr;
  static compartptr cmpt=NULL;
  static int inscan=0,count=0;

  if(inscan) goto scanportion;
  if(line2 && !strcmp(line2,"cmdtype")) return conditionalcmdtype(sim,cmd,4);

  cmptss=sim->cmptss;
  ... parsing of line2 ...
  cmpt=cmptss->cmptlist[c];

  count=0;
  inscan=1;
  molscan(sim,i,index,ms,cmd,cmdifincmpt);
  inscan=0;
  if((ch=='<' && count<min) || (ch=='=' && count==min) || (ch=='>' && count>min))
    return docommand(sim,cmd,line2);
  return CMDok;

 scanportion:
  mptr=(moleculeptr) line2;
  if(posincompart(sim,mptr->pos,cmpt)) count++;
  return CMDok; }
```

When the command is called initially, for running it, `inscan` equals 0 and `line2` is the line to be parsed, so control passes through the first two `if` statements. The command then parses `line2` and sets up the basic variables. Importantly, any variable that will be used with the individual molecules needs to be declared as a static variable. To scan through the molecules, the command sets `inscan` to 1 and runs the scan using `molscan`, while telling `molscan` to call back to itself with each individual molecule. It would be more straightforward if `molscan` called a different function, but that would require much more code and would also make it more difficult to pass information around, so that's why the control returns to the same function. When `molscan` calls back to this function, `inscan` is set to 1, so the flow jumps down to the `scanportion` label, where the individual molecule is processed. The molecule is sent to the command using the `line2` parameter, cast as a `char*`. After the molecule is processed, the command returns `CMDok` to indicate that the scan should continue. Finally, all molecules are done, `molscan` returns control back to the main portion of the command, and the command resets `inscan` back to 0 to indicate that the scan is done. It then finishes what it needs to do and returns `CMDok` to indicuate that the task is complete.

## Externally accessible function

Not all functions are listed here because many of them don't require any more description than what is already given in the Smoldyn User Manual.

`CMDcode docommand(void *cmdfnarg,cmdptr cmd,char *line);` `docommand` is given the simulation structure in `sim`, the command to be executed in `cmd`, and a line of text which includes the entire command string. It parses the line of text only into the first word, which specifies which command is to be run, and into the rest of the line, which contains the command parameters. The rest of the

line is then sent to the appropriate command routine as `line2`. The return value of the command that was called is passed back to the main program from `docommand`. These routines return `CMDok` for normal operation, `CMDwarn` for an error that does not require simulation termination, `CMDabort` for an error that requires immediate simulation termination, `CMDstop` for a normal simulation termination, and `CMDpause` for simulation pausing.

## Individual command functions

**simulation control**

`enum CMDcode cmdstop(simptr sim,cmdptr cmd,char *line2);`
Returns a value of 2, meaning that the simulation should stop. Any contents of `line2` are ignored.

`enum CMDcode cmdpause(simptr sim,cmdptr cmd,char *line2);`
Causes the simulation to pause until the user tells it to continue. Continuation is effected by either pressing the space bar, if OpenGL is used for graphics, or by pressing enter if output is text only. The return value is 0 for non-graphics and 3 for graphics. Any contents of `line2` are ignored.

`enum CMDcode cmdbeep(simptr sim,cmdptr cmd,char *line2);`
Causes the computer to beep (sent to stderr). Any contents of `line2` are ignored.

`enum CMDcode cmdkeypress(simptr sim,cmdptr cmd,char *line2);`
Sets a key press event as though the key had actually been pressed.

`enum CMDcode cmdsetflag(simptr sim,cmdptr cmd,char *line2);`
Sets the command superstructure flag value.

`enum CMDcode cmdsetrandseed(simptr sim,cmdptr cmd,char *line2);`
Sets the random number seed. Negative values indicate that the current time should be used.

`enum CMDcode cmdsetgraphics(simptr sim,cmdptr cmd,char *line2);`
Sets the type of graphics output.

`enum CMDcode cmdsetgraphic_iter(simptr sim,cmdptr cmd,char *line2);`
Sets the number of iterations between each graphics update.

**file manipulation**

`enum CMDcode cmdoverwrite(simptr sim,cmdptr cmd,char *line2);`
Overwrites a prior output file. See the user manual.

`enum CMDcode cmdincrementfile(simptr sim,cmdptr cmd,char *line2);`
Closes a file, increments the name and opens that one for output. See the user manual.

`enum CMDcode cmdsetrandseed(simptr sim,cmdptr cmd,char *line2);`
Sets the random number seed.

**conditional**

`enum CMDcode cmdifflag(simptr sim,cmdptr cmd,char *line2);`
Runs the command in `line2` depending on value of the command superstructure flag value.

`enum CMDcode cmdifprob(simptr sim,cmdptr cmd,char *line2);`
Runs the command in `line2` depending on a random number.

`enum CMDcode cmdifno(simptr sim,cmdptr cmd,char *line2);`
Reads the first word of `line2` for a molecule name and then checks the appropriate simulation live list to see if any molecules of that type exist. If so, it does nothing, but returns 0. If not, it sends the remainder of `line2` to `docommand` to be run as a new command, and then returns 0. It returns 1 if the molecule name was missing or not recognized.

`enum CMDcode cmdifless(simptr sim,cmdptr cmd,char *line2);`
Identical to `cmdifno`, except that it runs the command in `line2` if there are less than a listed number of a kind of molecules in the appropriate live list.

`enum CMDcode cmdifmore(simptr sim,cmdptr cmd,char *line2);`
Identical to `cmdifno` except that it runs the command in `line2` if there are more than a listed number of a kind of molecules in the appropriate live list.

`enum CMDcode cmdifincmpt(simptr sim,cmdptr cmd,char *line2);`
Runs a command depending on the number of molecules in a compartment.

`enum CMDcode cmdifchange(simptr sim,cmdptr cmd,char *line2);`
Runs a command if the number of some species of molecules changes, where the user can specify various changing criteria. This function uses `i1` to indicate whether the function has been called before and `i2` to store the number of molecules that were counted in the prior invocation.

`enum CMDcode cmdif(simptr sim,cmdptr cmd,char *line2);`
Runs a command if one value is greater than, less than, or equal to another value.

**observation commands**

`enum CMDcode cmdwarnescapee(simptr sim,cmdptr cmd,char *line2);`
Checks for molecules that escaped from the system and displays information about them.

`enum CMDcode cmdecho(simptr sim,cmdptr cmd,char *line2);`
Echos a string of text to the filename that is given.

`enum CMDcode cmdevaluate(simptr sim,cmdptr cmd,char *line2);`
Evalutates a math expression (usually a Smoldyn function) and prints the result to the filename that is given.

`enum CMDcode cmdmolcountheader(simptr sim,cmdptr cmd,char *line2);`
Prints a header line for the `molcount` collection of commands.

`enum CMDcode cmdmolcount(simptr sim,cmdptr cmd,char *line2);`
Reads the output file name from `line2`. Then, to this file, it saves one line of text listing the current simulation time, followed by the number of each type of molecule in the system. This routine does not affect any simulation parameters. It accounts for both particle space and lattice space.

`enum CMDcode cmdmolcountinbox(simptr sim,cmdptr cmd,char *line2);`
Counts and outputs the number of molecules in a specific box.

`enum CMDcode cmdmolcountincmpt(simptr sim,cmdptr cmd,char *line2);`
Counts and outputs the number of molecules in a specific compartment.

`enum CMDcode cmdmolcountincmpts(simptr sim,cmdptr cmd,char *line2);`
Counts and outputs the number of molecules in multiple compartments.

`enum CMDcode cmdmolcountincmpt2(simptr sim,cmdptr cmd,char *line2);`
Counts and outputs the number of molecules in a specific compartment.

`enum CMDcode cmdmolcountonsurf(simptr sim,cmdptr cmd,char *line2);`
Counts and outputs the number of molecules on a specific surface.

`enum CMDcode cmdmolcountspace(simptr sim,cmdptr cmd,char *line2);`
Counts and outputs the number of molecules along a line profile, creating a histogram.

`enum CMDcode cmdmolcountspace2d(simptr sim,cmdptr cmd,char *line2);`
Counts and outputs the number of molecules in a 2D grid, creating a 2D histogram.

`enum CMDcode cmdmolcountspaceradial(simptr sim,cmdptr cmd,char *line2);`
Counts and outputs the number of molecules as a function of radius, creating a histogram.

`enum CMDcode cmdmolcountpolarangle(simptr sim,cmdptr cmd,char *line2);`
Counts and outputs the number of molecules as a function of polar angle, creating a histogram.

`enum CMDcode cmdradialdistribution(simptr sim,cmdptr cmd,char *line2);`
Computes and outputs a radial distribution function.

`enum CMDcode cmdmolcountspecies(simptr sim,cmdptr cmd,char *line2);`
Counts and outputs the number of molecules of a given species.

`enum CMDcode cmdmolcountspecieslist(simptr sim,cmdptr cmd,char *line2);`
Counts and outputs the number of molecules of multiple species.

`enum CMDcode cmdlistmols(simptr sim,cmdptr cmd,char *line2);`
Reads the output file name from `line2`. To this file, it saves a list of every individual molecule in both live lists of the simulation, along with their positions. This routine does not affect any simulation parameters.

`enum CMDcode cmdlistmols2(simptr sim,cmdptr cmd,char *line2);`
Reads the output file name from `line2`. To this file, it saves the number of times this command was invoked using the invoke element of commands, a list of every individual molecule in both live lists of the simulation, along with their positions. This routine does not affect any simulation parameters. Routine originally written by Karen Lipkow and then rewritten by me.

`enum CMDcode cmdlistmols3(simptr sim,cmdptr cmd,char *line2);`
Reads a molecule name and the output file name from `line2`. To this file, it saves the number of times the command was invoked, the identity of the molecule specified, and the positions of every molecule of the specified type. This routine does not affect any simulation parameters.

`enum CMDcode cmdmolpos(simptr sim,cmdptr cmd,char *line2);`
Reads a molecule name and then the output file name from `line2`. To this file, it saves one line of text with the positions of each molecule of the listed identity. This routine does not affect any simulation parameters.

`enum CMDcode cmdtrackmol(simptr sim,cmdptr cmd,char *line2);`
Reads a molecule serial number and an output file name from `line2`. If the given serial number is large enough to have been assigned to a molecule, this searches all molecule lists for a molecule with that serial number. If it is found, this prints out the time, molecule species name, molecule state, and whether this molecule is inside or outside of each compartment sequentially. This function is primarily designed for debugging, but could be useful for single molecule tracking too. This routine does not affect any simulation parameters.

`enum CMDcode cmdmolmoments(simptr sim,cmdptr cmd,char *line2);`
Reads a molecule name and then the output file name from `line2`. To this file, it saves in one line of text: the time and the zeroth, first, and second moments of the distribution of positions for all molecules of the type listed. The zeroth moment is just the number of molecules (of the proper identity); the first moment is a dim dimensional vector for the mean position; and the second moment is a dimxdim matrix of variances. This routine does not affect any simulation parameters.

`enum CMDcode cmdsavesim(simptr sim,cmdptr cmd,char *line2);`
Reads the output file name from `line2` and then saves the complete state of the system to this file, as a configuration file. This output can be run later on to continue the simulation from the point where it was saved.

`void cmdmeansqrdispfree(cmdptr cmd);`
A memory freeing routine for memory that is allocated by `cmdmeansqrdisp`.

`enum CMDcode cmdmeansqrdisp(simptr sim,cmdptr cmd,char *line2);`
This calculates the mean square displacements of all molecules of the requested type, based on the difference between their current positions and their positions when the command was first invoked. This uses several of the command memory storage options. `i1` is the number of molecules being tracked and is the size of other arrays; `i2` is 0 if memory and initial values have not been set up, 1 if they have, or 2 if the function failed; `v1` is the list of molecule serial numbers; and `v2` is the list of initial coordinates for each molecule.

`enum CMDcode cmdmeansqrdisp2(simptr sim,cmdptr cmd,char *line2);`
This calculates the mean square displacements of molecules of the requested type, based on the difference between their current or latest positions and their positions when they were first tracked. This uses several of the command memory storage options. `i1` is the maximum number of molecules that can be tracked and is the size of other arrays; `i2` is 0 if memory and initial values have not been set up, 1 if they have, or 2 if the function failed; `i3` is the actual number of molecules being tracked; `v1` is the list of molecule serial numbers; `v2[0]` is a code equal to 0 if the molecule is not being tracked, 1 if it's not being tracked and exists, 2 if it's being tracked, or 3 if it's being tracked and exists; `v2[1,..,dim]` is the list of initial coordinates for each molecule; and `v2[dim+1,..,2*dim]` is the list of current coordinates for each molecule.

`enum CMDcode cmdmeansqrdisp3(simptr sim,cmdptr cmd,char *line2);`
This calculates the effective diffusion coefficient of molecules of the requested type, based on the difference between their current or latest positions and their positions when they were first tracked, as well as their diffusion times. Effective diffusion coefficients are computed by weighting in direct proportion to the molecule lifetime. To do this, effective diffusion coefficients are the sum of squared displacements divided by (2 times the dimensionality times the sum of the molecule lifetimes). This function was modified from `cmdmeansqrdisp2`. This uses several of the command memory storage options. `i1` is the maximum number of molecules that can be tracked and is the size of other arrays; `i2` is 0 if memory and initial values have not been set up, 1 if they have, or 2 if the function failed; `i3` is the actual number of molecules being tracked; `v1` is the list of molecule serial numbers; `v2[0]` is a code equal to 0 if the molecule is not being tracked, 1 if it's not being tracked and exists, 2 if it's being tracked, or 3 if it's being tracked and exists; `v2[1,..,dim]` is the list of initial coordinates for each molecule; `v2[dim+1,..,2*dim]` is the list of current coordinates for each molecule, `v2[2*dim+1]` is the simulation time when tracking began on this molecule; and `f1` is the effective diffusion coefficient from the prior invocation, or -1 on initialization.

`enum CMDcode cmdresidencetime(simptr sim,cmdptr cmd,char *line2);`
This calculates the residence time of molecules of the requested type, based on the difference between the current time and the molecule creation times. This function was modified from `cmdmeansqrdisp3`. This uses several of the command memory storage options. `i1` is the maximum number of molecules that can be tracked and is the size of other arrays; `i2` is 0 if memory and initial values have not been set up, 1 if they have, or 2 if the function failed; `i3` is the actual number of molecules being tracked; `v1` is the list of molecule serial numbers; `v2[0]` is a code equal to 0 if the molecule is not being tracked, 1 if it's not being tracked and exists, 2 if it's being tracked, or 3 if it's being tracked and exists; and `v2[1]` is the simulation time when tracking began on this molecule.

`enum CMDcode cmddiagnostics(simptr sim,cmdptr cmd,char *line2);`
Displays diagnostics about different data structures to the standard output.

`enum CMDcode cmdexecutiontime(simptr sim,cmdptr cmd,char *line2);`
Prints simulation time and execution time to file.

`enum CMDcode cmdprintLattice(simptr sim,cmdptr cmd,char *line2);`
Prints information about the lattices to the a file. This would be better as part of the cmddiagnostics command.

`enum CMDcode cmdwriteVTK(simptr sim,cmdptr cmd,char *line2);`
Outputs VTK data for the current simulation state.

**system manipulation**

`enum CMDcode cmdset(simptr sim,cmdptr cmd,char *line2);`
Reads `line2` to extract the first word and the rest of the line. These are passed to `simreadstring` where they are interpreted as a configuration file line. Any errors are passed on from `simreadstring`.

`enum CMDcode cmdpointsource(simptr sim,cmdptr cmd,char *line2);`
Reads `line2` for a molecule name, followed by the number of molecules that should be created, followed by the `dim` dimensional position for them. If all reads well, it creates the new molecules in the system at the appropriate position.

`enum CMDcode cmdvolumesource(simptr sim,cmdptr cmd,char *line2);`
Reads `line2` for a molecule name, the number of molecules that should be created, and a region. If all reads well, it creates the new molecules in the system in the appropriate volume.

`enum CMDcode cmdgaussiansource(simptr sim,cmdptr cmd,char *line2);`
Reads `line2` for a molecule name, the number of molecules that should be created, and a region. If all reads well, it creates the new molecules in the system with a Gaussian distribution.

`enum CMDcode cmdmovesurfacemol(simptr sim,cmdptr cmd,char *line2);`
Probabilistically moves molecules from one surface to another, with an optional state change.

`enum CMDcode cmdkillmolinsphere(simptr sim,cmdptr cmd,char *line2);`
Reads `line2` for a molecule name and a surface name and then kills all molecules of the given type, that are in spheres of the listed surface. The molecule name and/or the surface name can be "all".

`enum CMDcode cmdkillmolincmpt(simptr sim,cmdptr cmd,char *line2);`
Reads `line2` for a molecule name and a compartment name and then kills all molecules of the given type, that are in the listed compartment. The molecule name can be "all".

`enum CMDcode cmdkillmoloutsidesystem(simptr sim,cmdptr cmd,char *line2);`
Reads `line2` for a molecule name and then kills all molecules of the given type that are outside of the system boundaries. The molecule name and/or the surface name can be "all".

`enum CMDcode cmdfixmolcount(simptr sim,cmdptr cmd,char *line2);`
Fixes the copy number of a specific molecule species, only considering solution phase, to a specified quantity. This considers the entire system, and adds or removes molecules as required.

`enum CMDcode cmdfixmolcountrange(simptr sim,cmdptr cmd,char *line2);`
Fixes the copy number of a specific molecule species, only considering solution phase, to a specified range. This considers the entire system, and adds or removes molecules as required.

`enum CMDcode cmdfixmolcountonsurf(simptr sim,cmdptr cmd,char *line2);`
Fixes the copy number of surface-bound molecules to a specified value. This considers an entire surface, and adds or removes molecules as required.

`enum CMDcode cmdfixmolcountonsurf(simptr sim,cmdptr cmd,char *line2);`
Fixes the copy number of surface-bound molecules to a specified range. This considers an entire surface, and adds or removes molecules as required.

`enum CMDcode cmdfixmolcountincmpt(simptr sim,cmdptr cmd,char *line2);`
Fixes the copy number of molecules in a compartment to a specified value. This adds or removes molecules as required.

`enum CMDcode cmdfixmolcountrangeincmpt(simptr sim,cmdptr cmd,char *line2);`
Fixes the copy number of molecules in a compartment so that it is within the specified range. This adds or removes molecules as required.

`enum CMDcode cmdequilmol(simptr sim,cmdptr cmd,char *line2);`
Equilibrates a pair of molecular species, allowing the efficient simulation of rapid reactions. It reads two molecule names from `line2`, followed by a probability value. Then, it looks for all molecules in the live lists with either of the two types and replaces them with the second type using the listed probability or with the first type using 1 the listed probability.

`enum CMDcode cmdreplacexyzmol(simptr sim,cmdptr cmd,char *line2);`
Reads the name of a molecule following by a `dim` dimensional point in space from `line2`. Then, it searches the fixed live list for any molecule that is exactly at the designated point. If it encounters one, it is replaced by the listed molecule, and then the live lists are sorted if appropriate. This routine stops searching after one molecule has been found, and so will miss additional molecules that are at the same point.

`enum CMDcode cmdreplacevolmol(simptr sim,cmdptr cmd,char *line2);`
Replaces one molecule species in a volume with another, with some replacement probability.

`enum CMDcode cmdreplacecmptmol(simptr sim,cmdptr cmd,char *line2);`
Replaces one molecule species in a compartment with another, with some replacement probability. This is a modification of `cmdreplacevolmol`, and uses a little code from `cmdfixmolcountincmpt`.

`enum CMDcode cmdmodulatemol(simptr sim,cmdptr cmd,char *line2);`
Identical to `cmdequilmol` except that the equilibration probability is not fixed, but is a sinusoidally varying function. After reading two molecule names from `line2`, this routine then reads the cosine wave frequency and phase shift, then calculates the probability using the function $prob = 0.5 * (1.0 - cos(freq * \texttt{sim->time} + shift))$.

`enum CMDcode cmdreact1(simptr sim,cmdptr cmd,char *line2);`
Reads `line2` for the name of a molecule followed by the name of a unimolecular reaction. Then, every one of that type of molecule is caused to undergo the listed reaction, thus replacing each one by reaction products. Molecules are sorted at the end. This might be useful for simulating a pulse of actinic light, for example.

`enum CMDcode cmdsetrateint(simptr sim,cmdptr cmd,char *line2);`
This reads `line2` for the name of a reaction and the new internal rate constant for it. The internal rate constant is set to the new value. Errors can arise from illegal inputs, such as the reaction not being found or a negative requested internal rate constant.

`enum CMDcode cmdshufflemollist(simptr sim,cmdptr cmd,char *line2);`
Shuffles one or more of the live internal molecule lists.

`enum CMDcode cmdshufflereactions(simptr sim,cmdptr cmd,char *line2);`
Shuffles reactions for one or more reactant combinations, for bimolecular reactions. This function is somewhat inefficient in that it shuffles all lists twice if it is called for all reactant pairs, but I decided to keep the inefficiency because improvement, and still allowing for scanning over all of either reactant individually, would lead to slower and messier code.

`enum CMDcode cmdsetsurfcoeff(simptr sim,cmdptr cmd,char *line2);`
Sets the surface interaction rate and then calls `surfacesupdate` to update the probabilities. Zsuzsanna Sukosd wrote this command.

`enum CMDcode cmdsettimestep(simptr sim,cmdptr cmd,char *line2);`
This reads `line2` for the new simulation time step. Nothing else is changed in the simulation, including binding or unbinding radii, so reaction rates may be observed to change.

`enum CMDcode cmdexcludebox(simptr sim,cmdptr cmd,char *line2);`
Allows a region of the simulation volume to be effectively closed off to molecules. The box is defined by its low and high corners, which are read from `line2`. Any molecule, of any type, that entered the box during the last time step, as determined by its `pos` and `posx` structure members, is moved

back to its previous position. This is not the correct behavior for a reflective surface, but is efficient and expected to be reasonably accurate for most situations. This routine ought to be replaced with a proper treatment of surfaces in the main program (rather than with interpreter commands), but that's a lot more difficult.

`enum CMDcode cmdexcludesphere(simptr sim,cmdptr cmd,char *line2);`
Like `cmdexcludebox` except that it excludes a sphere rather than a box. The sphere is defined by its center and radius, which are read from `line2`. Any molecule, of any type, that entered the sphere during the last time step, as determined by its `pos` and `posx` structure members, is moved back to its previous position. This is not the correct behavior for a reflective surface, but is efficient and expected to be reasonably accurate for most situations.

`enum CMDcode cmdincludeecoli(simptr sim,cmdptr cmd,char *line2);`
This is the opposite of the `excludebox` and `excludesphere` commands. Here, molecules are confined to an *E. coli* shape and are put back inside it if they leave. See the user manual for more about it. Unlike the other rejection method commands, this one works even if a molecule was in a forbidden region during the previous time step; in this case, the molecule is moved to the point on the *E. coli* surface that is closest. Because of this difference, this command works reasonably well even if it is not called at every time step.

`enum CMDcode cmdsetreactionratemolcount(simptr sim,cmdptr cmd,char *line2);`
This command sets the reaction rate to be a linear function of one or more molecule counts. I considered letting the user select compartments for the molecules, but that turned out to be too much work. Instead, I suggest that the user use different species names in different regions of space as much as convenient, and use wildcards to collect different species together. That will run faster anyhow.

`enum CMDcode cmdexpandsystem(simptr sim,cmdptr cmd,char *line2);`
This command expands or contracts the entire system, including all molecule positions and surfaces. The work for surfaces is performed by `surftransformpanel`.

`enum CMDcode cmdtranslatecmpt(simptr sim,cmdptr cmd,char *line2);`
This command translates a compartment, including dealing with molecule displacements. All of the work is done by `cmpttranslate`.

`enum CMDcode cmddiffusecmpt(simptr sim,cmdptr cmd,char *line2);`
This command diffuses a compartment, including dealing with molecule displacements. All of the work is done by `cmpttranslate`.

**Internal functions**

`void cmdv1free(cmdptr cmd);`
Frees array `cmd->v1`.

`void cmdv1v2free(cmdptr cmd);`
Frees arrays `cmd->v1` and `cmd->v2`.

`enum CMDcode conditionalcmdtype(simptr sim,cmdptr cmd,int nparam);`
Returns the command type for conditional commands, which are required to return the type of the function that gets called if the condition is true. `cmd` is the conditional command and `nparam` is the number of parameters for the conditional command (e.g. for `cmdifno`, the only parameter is the molecule name, so `nparam` is 1).

`int insideecoli(double *pos,double *ofst,double rad,double length);`
This is a short utility routine used by the command `cmdincludeecoli`. It returns a 1 if a molecule is inside an *E. coli* shape and a 0 if not. `pos` is the molecule position, `ofst` is the physical location of the cell membrane at the center of the low end of the cell (the cell is assumed to have its long axis along the $x$-axis), `rad` is the cell radius used for both the cylindrical body and the hemispherical ends, and length is the total cell length, including both hemispherical ends.

`void putinecoli(double *pos,double *ofst,double rad,double length);`
This is another short utility routine used by the command `cmdincludeecoli`. It moves a molecule from its initial position in `pos` to the nearest surface of an *E. coli* shape. Parameters are the same as those for `insideecoli`.

`int molinpanels(simptr sim,int ll,int m,int s,char pshape);`
This function, which might be better off in the smolsurf.c code, is used to test if molecule number `m` of live list `ll` is inside any of the `pshape` panels of surface number `s`. Only spheres are allowed currently as panel shapes, because neither rectangles nor triangles can contain molecules. If `s` is sent in with a value less than 0, this means that all `pshape` panels of all surfaces will be checked.

## 4.17 Top-level code (functions in smoldyn.c)

The top-level source code file, smoldyn.c, contains only the `main` function for the stand-alone Smoldyn program. It is declared locally in this source file and so cannot be called from externally, except from the shell. This source code file is not included in Libsmoldyn.

`int main(int argc,char *argv[]);`
`main` is a simple routine that provides an entry point to the program from the shell. It checks the command line arguments, prints a greeting, inputs the configuration file name from the user, and then calls `setupsim` to load the configuration file and set up all the structures. If all goes well, it calls `simulate` or `simulategl` to run the simulation. At the end, it returns to the shell.

# Chapter 5

# Code design

This chapter describes interactions between different portions of the code. The code is, and may always be, in flux. While I try to maintain this section of the documentation, be forewarned that it might not reflect the most recent changes.

## 5.1 Memory management

The following table, *which is very out of date* shows memory allocation and freeing for the different structures. For both the allocation and freeing columns, the top line shows the function in which the structure is actually allocated or freed, while subsequent lines show the functions that call the preceding functions.

| structure | allocation | freeing |
|---|---|---|
| moleculestruct | molalloc | molfree |
| | molexpandlist | molssfree |
| | molsetmaxmol, molsort | simfree |
| | simreadstring (max_mol), ? | |
| | loadsim, ? | |
| | setupsim, ? | |
| moleculesuperstruct | molssalloc | molssfree |
| | molsetmaxspecies | simfree |
| | simreadstring (max_species,..) | |
| | loadsim | |
| | setupsim | |
| | molssetgausstable | |
| | simreadstring (gauss_table_size), setupmols | |
| | loadsim, simupdate | |
| | setupsim; setupsim, simulatetimestep | |
| | mollistalloc | |
| | addmollist | |
| | setupmols, simreadstring (molecule_lists), setupports | |
| | simupdate, loadsim | |
| | setupsim | |
| wallstruct | wallalloc | wallfree |
| | wallsalloc | wallsfree |
| | walladd | simfree |
| | simreadstring (boundaries,..) | |

| | | |
|---|---|---|
| | loadsim | |
| | setupsim | |
| rxnstruct | rxnalloc | rxnfree |
| | RxnAddReaction | rxnssfree |
| | loadsim (reaction,..) | simfree |
| | setupsim | |
| rxnsuperstruct | rxnssalloc | rxnssfree |
| | RxnAddReaction | simfree |
| | simreadstring (reaction,..), loadrxn | |
| | loadsim, ? | |
| | setupsim | |
| panelstruct | panelsalloc | panelfree |
| | surfreadstring (max_panels) | surfacefree |
| | loadsurface | surfacessfree |
| | simreadstring (start_surface) | simfree |
| | loadsim | |
| | setupsim | |
| surfacestruct | surfacealloc | surfacefree |
| | surfacessalloc | surfacessfree |
| | simreadstring (max_surface) | simfree |
| | loadsim | |
| | setupsim | |
| surfacesuperstruct | surfacessalloc | surfacessfree |
| | surfenablesurfaces | simfree |
| | simreadstring (max_surface) | |
| | loadsim | |
| | setupsim | |
| boxstruct | boxalloc | boxfree |
| | boxesalloc | boxesfree |
| | setupboxes | boxssfree |
| | simupdate | simfree |
| | setupsim | |
| boxsuperstruct | boxssalloc | boxssfree |
| | boxsetsize | simfree |
| | simreadstring (boxsize,..), setupboxes | |
| | loadsim, simupdate | |
| | setupsim, setupsim | |
| compartstruct | compartalloc | compartfree |
| | compartssalloc | compartssfree |
| | simreadstring (max_compartment) | simfree |
| | loadsim | |
| | setupsim | |
| compartsuperstruct | compartssalloc | compartssfree |
| | simreadstring (max_compartment) | simfree |
| | loadsim | |

| | | |
|---|---|---|
| | setupsim | |
| portstruct | portalloc | portfree |
| | portssalloc | portssfree |
| | simreadstring (max_port) | simfree |
| | loadsim | |
| | setupsim | |
| portsuperstruct | portssalloc | portssfree |
| | simreadstring (max_port) | simfree |
| | loadsim | |
| | setupsim | |
| mzrsuperstruct | mzrssalloc | mzrssfree |
| | mzrssload | simfree |
| | simreadstring (read_network_rules) | |
| | loadsim | |
| | setupsim | |
| simstruct | simalloc | simfree |
| | setupsim | |

## 5.2 Data structure preparation and updating

The original Smoldyn design was that it read a configuration file, set up internal data structures, ran the simulation, and then quit. Two problems arose. First, the sequence for setting up the internal data structures became increasingly complicated as the program gained features, and second, this overall program flow wasn't always what users wanted. As a result, the code is now more modularized and it includes a "condition" element in each data structure that reports on the overall state of that data structure. These help, but data structure set up and updating are still somewhat complicated, which is the focus of this section.

Note that the condition element and the data structure updating discussed here are designed for relatively infrequent system modifications. That is, they are for changes to the system, performed by the user or by run-time commands. They are not designed for regular simulation operation, such as for updates after molecules diffuse or reactions occur.

The highest level updating function is `simupdate`. This can be called with data structures that have just been loaded in from a configuration file, that have just been created using Libsmoldyn functions, that have been modified using run-time commands, or that are in good running order. It will take care of all necessary updates. `simupdate` calls the other update functions in the order listed below. Because updates cannot be finished until others are started, `simupdate` will sometimes call update functions multiple times, until everything is done.

All update functions return the same types of values. They return 0 for success, 1 if memory could not be allocated, 2 if a different data structure that needed updating hasn't been adequately updated yet, or larger numbers for other errors.

Structure conditions (SC) are divided into four categories: `SCinit`, `SClists`, `SCparams`, and `SCok`; these are enumerated in the `StructCond` enumerated list. `SCinit` implies that a structure is still being initialized, `SClists` implies that one or more of the lists that comprise a structure need updating, `SCparams` implies that one or more of the structure simulation parameters need updating, and `SCok` implies that the structure is fully updated and ready for use. The items that are initialized for each condition are shown below. This table was updated after changes for version 2.23.

`molsupdate`
- `SClists (molsupdatelists)`
  - gaussian lookup table
  - `mols->exist` (calls `rxnisprod` and `issurfprod`)

creates system molecule lists if none yet
sets any list lookup values that weren't done yet
sets molecule list values for molecules in dead list

`SCparams (molsupdateparams)`
`mols->difstep`
`mols->diffuselist`

`boxesupdate`

`SClists (boxesupdatelists)`
box superstructure (requires **walls**)
`bptr->indx`
neighbor boxes
`bptr->nwall` and `bptr->wlist`

`SCparams (boxesupdateparams)`
`bptr->npanel` and `bptr->panel` (requires **surfaces**)
molecules in boxes (requires **molecules** and assigns `mptr->box`)

`molsort`

resets `topl` indices
sorts and compacts live lists (calls `boxremovemol` and `boxaddmol`)
moves molecules from resurrected to reborn lists (calls `boxaddmol`)
resets `sortl` indices

`compartsupdate`

`SClists (compartsupdatelists)`
does nothing

`SCparams (compartsupdateparams)`
finds boxes in compartment (requires **boxes** and **surfaces**)
finds box volumes and compartment volumes

`rxnsupdate`

`SClists (rxnsupdatelists)`
sets reaction molecule lists (requires **molecule lists**)

`SCparams (rxnsupdateparams)`
sets rates (may require **compartments**)
sets products
calculates tau values

`surfupdate`

`SClists (surfupdatelists)`
allocates `srfmollist` arrays (requires **molecule lists**)
sets `srfmollist` values (requires **molecule lists** and calls `rxnisprod`)

`SCparams (surfupdateparams)`
sets surface interaction probabilities (requires `mols->difc` and `->difstep`)

`portsupdate`

`SClists (portsupdatelists)`
sets port molecule lists (uses **molecule lists**)

`SCparams (portsupdateparams)`
does nothing

## 5.3 Simulation algorithm sequence

In a sense, the core function of the entire Smoldyn program is `simulatetimestep`, which is in smolsim.c. Using the assumption that all data structures are given to this function are in good working order, `simulatetimestep` runs the simulation for one time step, and then cleans up all of the data structures as required, thus leaving them again in good working order. The sequence of tasks performed here is central to accurate and efficient simulation, as well as data structure maintenance.

As mentioned in the Smoldyn User Manual, the simulation concept is that the system can be observed at a fixed time, then evolves to a new state, can be observed again, and so forth. The details of what happens during the time evolution should not be relevant to the user. Instead, all that should matter is that all aspects of the observable simulated states conform as closely as possible to results that would be found from the ideal and hypothetical continuous-time model system that the simulation is intended to represent. Furthermore, the simulation results should be as independent of the time step as possible, and they should converge to those of the hypothetical model system as the time step is reduced towards zero.

Here, we focus on the details of what happens during a time step. The sequence of simulation algorithms is:

| operation | error code | box assignment | molecule sorting | surface sides |
|---|---|---|---|---|
| ___time = $t$___ | | | | |
| observe and manipulate system | er | ok | bad | ok |
| sort molecule lists | n/a | ok | ok | ok |
| graphics are drawn | n/a | ok | ok | ok |
| ___start/end of `simulatetimestep`___ | | ok | ok | ok |
| **diffusion** | | | | |
| molecules diffuse | 9 | bad | ok | bad |
| **surface collisions** | | | | |
| surface collision interactions | n/a | bad | bad | ok |
| **reactions** | | | | |
| desorption and surface-state transitions | 6 | bad | bad | ok |
| assign molecules to boxes | 2 | ok | bad | ok |
| 0th order reactions | 3 | ok | bad | bad |
| 1st order reactions | 4 | ok | bad | bad |
| 2nd order reactions | 5 | ok | bad | bad |
| sort molecule lists | 6 | ok | ok | bad |
| surface collisions again | n/a | bad | ok | ok |
| assign again | 2 | ok | ok | ok |
| ___time = $t + \Delta t$___ | | | | |

The overall picture is that the simulation sequence is: diffusion, surface interactions, and reactions. The "desorption and surface-state transitions" operation is really a reaction, and so it is debatable whether it should be before or after chemical reactions. I chose to put it where it is because it shares overlap with surface collision interactions, which have to be before reactions.

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).

## 5.4 Wildcards, species groups, and patterns

### Patterns

Most of Smoldyn's treatment of wildcards, species groups, and patterns is in the molecules and reactions portion of the code. The basic idea is that any string that the user enters for a species is treated as a "pattern". This pattern might represent one species or multiple species.

Patterns are stored in the molecule superstructure. The `PatternData` enumeration, of which there are `PDMAX` options, are for the first `PDMAX` elements of the `mols->patindex` array, called the header. This array, as explained below, lists the species indices that match to text patterns. Each listing requires some information to describe how long the listing is, what's stored in it, and what it corresponds to, which is included in the header. These header elements should be retreived using the enumerated values. See below for their description.

Patterns are strings, each of which represents one or more species names. Each string that is used gets recorded as a pattern; however, those that are not used (e.g. species that were generated automatically but never referenced individually by the user) are not recorded here, so there is no certainty that the list here is complete. This list is only updated when it is used.

Patterns are stored in the molecule superstructure. This paragraph describes how they are stored. `maxpattern` slots are allocated for patterns, of which `npattern` are actually used. The actual list of patterns is called `patlist`; these are sorted by alphabetical order. Sorted in parallel are the lists `patindex` which is a list of lists, and `patrname`, which is a list of reaction names. `patrname` is only used if the pattern represents a reaction, and is used to differentiate between multiple different reactions that have identical patterns. In `patindex`, there is one element for each pattern. Each element of `patindex` is a sublist.

The contents of each `patindex` list starts with a header which gives important information about the list. The header occupies the first `PDMAX` list elements. They are:

| element | name | meaning |
|---|---|---|
| 0 | PDalloc | allocated sublist size including header values |
| 1 | PDnresults | number of stored results |
| 2 | PDnspecies | number of species during last sublist update |
| 3 | PDmatch | number of "match species" in this pattern |
| 4 | PDsubst | number of "replace species" in this pattern |
| 5 | PDrule | 0 if this is not a "rule" and 1 if it is (only rules generate new species) |

`PDalloc` is the total size of the index list. This value is checked when the index list needs to be expanded and is updated exclusively by `molpatternindexalloc`. `PDnresults` is the number of stored results, not including the values in the header. It may be 0 or more. `PDnspecies` is the number of species that existed in the system when the index list was last updated. This value is initialized to 1, because a minimal simulation that includes molecules includes an empty species. It is increased to `sim->mols->nspecies` at each update. This value can also be set to -1 to indicate that it should not be updated in the future, even if more species are created (appropriate for a pattern that represents only a single species).

The number of spaces used for each stored result is the sum of the match and replace counts. For example, suppose a pattern is `A*\nX*`, which has been found to match and substitute to `AB\nXB`, `AC\nXC`, and `AD\nXD`. If the sublist were allocated with 20 elements, then element 0 would equal 13, element 1 would equal 3, element 2 would equal 6 if `nspecies` was 6 when this sublist was created, element 3 would equal 1 because there's one match word here, element 4 would equal 1 because there's one substitution word here, element 5 would equal 0 if this is not a rule, elements 6 and 7 would be the indices for the "AB" and "XB" species, elements 8 and 9 for "AC" and "XC" species, elements 10 and 11 for "AD" and "XD" species, and elements

12 to 19 would be empty. The list of results is sorted in ascending order.

## Control flow and functions - species names

The entry point for most functions that deal with species names that are provided by the user is `molstring2index1`. This function takes a string (typically a species name, a species group name, or a species name with wildcards, optionally followed by the state) and returns an index variable for the species, creating a new index if needed. It also returns the state, returning `MSsoln` if the string did not include a state. For example, adding a species group happens in `moladdspeciesgroup`; the first thing this does is to send the species group name to `molstring2index1` so that it can get back an index variable for the species group. The same function also sends its species name to `molstring2index1` so that it can get back an index variable for the species. `molstring2index1` is also called by `simreadstring` for essentially all input lines that include species, including "difc", "difm", "drift", "surface_drift", "mol", "surface_mol", "compartment_mol", "mol_list", "display_size", "color", and "product_placement". It is also called by a vast number of commands.

Internally, `molstring2index1` sends the string off to `molstring2pattern` where the string is separated into a pattern portion, which might be a species name, a species group name, or a species name with wildcards, and a state for the species. This function can also assemble more complex patterns if used with non-zero mode values, which are for reactions, described below. No wildcard stuff is parsed in `molstring2pattern`. This function simply returns the pattern and the state. Next, `molstring2index1` sends the pattern off to `molpatternindex` which looks to see if the pattern has been exapanded before into a list of species and, if so, returns that list of species. If not, it performs any necessary wildcard matching to create a list of species, which it both stores and returns in `index`. At the end, `molstring2index1` returns the index variable.

## Reaction rule storage

Reaction rules are stored within the reaction superstructure. A simulation can have up to three reaction superstructures, corresponding to order 0, 1, and 2 reactions, but it's best if all of the rules are stored together because then it's possible to keep them in the order that the user entered them. As a result, Smoldyn first looks for existing rules and if it doesn't find any, then it stores the rules in the lowest order superstructure that is currently defined (this happens at the top of `RuleAddRule`). Any functions that use rules need to look through all of the superstructures to find them.

The rule list, within the reaction superstructure, has allocated size `maxrule`, actual size `nrule` and several lists, which are `rulename`, `rulepattern`, `ruledetails`, and `rulerate`. The `rulename` list contains strings with the rule names, which are simply the reaction names entered by the user, without any suffix. The `rulepattern` list is a list of patterns which serve as reaction rules; a pattern for a bimolecular reaction has a single-species pattern for the first reactant, which is a species name possibly with wildcard characters, a space, a second single-species pattern, a newline character, and then space-separated product species names. The `ruledetails` list contains arrays of integers, each of which lists the species states cast as integers for the reactants and products in their sequence, then the compartment number if the reaction is restricted to a specific compartment, and then the surface number if the reaction is restricted to a single surface. The `rulerate` list contains the reaction rate entered by the user. The sole difference between reaction rules and reactions (which are not stored here) is that the rules are allowed to generate new species whereas reactions are not. Reaction rules are typically only useful in combination with wildcard characters. These lists are maintained by the `RuleAddRule` function. These rules do not need updating during the simulation. Instead, they are simply stored here and referred to when updating is required. The `ruleonthefly` element is initialized to 0. It is then set to 1 if on-the-fly rule generation is desired.

## Control flow and functions - reactions

Reactions are initially added to a simulation when the `simreadstring` function encounters a "reaction" or "reaction_rule" word in a configuration file. If that happens, the remainder of the line of input file text gets sent off to `rxnparsereaction` for processing. This function takes care of parsing that is usually done within

simreadstring but got so complicated that it got separated into its own function. That function reads the reaction name, the names of any compartment or surface that the reaction is restricted to, the reactant names and states, and the product names and states. As it reads the reactants and products, these get appended to pattern using the molstring2pattern function. This then calls RxnAddReactionPattern with the reaction name, the pattern, the reactant and product states, any compartment or surface restrictions, and whether this is a rule. Note that the reaction rate is not read or used so far.

RxnAddReactionPattern can be called either when a reaction is first created or during reaction expansion, so it first calls molpatternindex without updating to find the current pattern status. If that function does not return an index variable, which we'll assume in this case, that means the pattern is new and an index variable needs to be created. The function then calls molpatternindex a second time, now with updating, which creates the index variable. The index variable includes a list all of the species combinations, given the current species list, that agree with the given pattern. If the reaction does not include wildcards, index will list only a single reaction; vice versa, if it does include wildcards, then index may list multiple reactions. If the reaction is a rule, then molpatternindex will create any species that the reaction rule generates and will add those species to the simulation. Next, RxnAddReactionPattern loops over all of the new reactions in the index variable, which is all of them in this case, extracts the species identities, tests to see whether an identical reaction already exists, and calls RxnAddReaction to add the reaction if not. If the reaction does exist, then this increases the reaction multiplicity.

RxnAddReaction adds a new reaction to the simulation (it also updates an existing reaction if it was only entered with reactants previously). It takes in the reaction name, species, states, and compartment and surface restrictions, creates the appropriate reaction superstructure if needed, creates the requested reaction, adds that reaction to the superstructure, and downgrades the reaction and surface conditions to the SClists levels. This triggers reaction molecule list, rate, and product placement computations when updating is next performed. The surface list is here because products with non-zero unbinding radii can cross surfaces, so surface checking for this needs to be turned on.

After all of the reactions for the given pattern have been added, control returns to rxnparsereaction, which then reads the reaction rate, if it was entered. It sends this rate off to RxnSetValue if the pattern corresponded to only one reaction and to RxnSetValuePattern if the pattern corresponded to multiple reactions. In the latter case, RxnSetValuePattern simply calls molpatternindex without updating to get the index variable, then goes through the specific reactions in it and calls RxnSetValue for each of them. Importantly, RxnSetValuePattern checks to see if the reaction is being given a rate for the first time, setting the rate to the given value if so, and otherwise it adds the given value to the current reaction rate. This works because this function is only called during setup of multiple reactions or during expansion of rules. Finally, if the reaction is a rule, then rxnparsereaction calls RuleAddRule to store the rule for later use.

After this, Smoldyn continues on to read more lines from the input file, possibly repeating the same procedure if another reaction is entered.

## Control and flow - network generation

Reaction rule expansion occurs through the RuleExpandRules function. This function is called if the user uses the "expand_rules" statement, which is parsed by simreadstring. It is also called at every time step in the main loop, which is in the simulatetimestep function.

The RuleExpandRules function first checks to see if expansion is necessary. If so, it goes through the rules list sequentially. For each rule, it calls molpatternindex without updating to convert the pattern to an index, it gets the data it needs from the other rule lists, and then it calls RxnAddReactionPattern with this information. That function updates the index variable for this pattern and adds any new reactions. After control returns to RuleExpandRules, it calls RxnSetValuePattern along with the same information and also the starting point for updating, and that function sets the reaction rates for the new patterns, adding the rate to the existing rate.

Within the molecule superstructure, expand is a flag for on-the-fly rule-based modeling. It is initialized to 0 and stays that way so long as there have never been any molecules of this species, it is increased to 1 if at least one molecule of this species has been created but it has not yet been used for rule expansion, and is set to either 2 or 3 if it has been used for expansion.

# Chapter 6

# Smoldyn modifications

### Modifications for version 1.5 (released 7/03)

- Added hierarchical configuration file name support.
- `Zeroreact` assigns the correct box for new molecules.
- The user can choose the level of detail for the bimolecular interactions (just local, nearest neighbor, all neighbor, including periodic, etc.)
- Bimolecular reactions were slow if most boxes are empty. Solution was to go down molecule list rather than box list.
- Absorbing wall probabilities were made correct to yield accurate absorption dynamics at walls.
- Cleaned up and got rid of old commands.
- The current time input was made useful.
- Graphics were improved by adding perspective and better user manipulation.
- Simulation pausing was made possible using graphics and improved without graphics.
- If a command was used with a wrong file name, the command string became corrupted during the final command call. This was fixed by Steve Lay.
- Fixed the neighbor list for bimolecular reactions between mobile and immobile reactants.
- Reactions were made possible around periodic boundaries.
- Molecules were lost sometimes. This bug was fixed: 4 lines before end of `molsort`: was `while(!live[m])`, is now `while(!live[m]&&m<nl[ll])`.
- Output files now allow the configuration file to be in a different folder as Smoldyn.
- Added an output file root parameter.
- Added the command `replacexyzmol`. Afterwards, the code for the command was sped up considerably.
- Sped up the command `excludebox`.
- Command time reports were fixed for type `b` and `a` commands.
- Added more types of command timing codes.
- Improved accuracy of `unireact` so that it correctly accounts for multiple reactions from one identity.
- Improved product parameter entry and calculation, as well as the output about reaction parameters.
- Added the routine `checkparams` to check that the simulation parameters are reasonable.

## Modifications for version 1.51 (released 9/5/03)

- Fixed a minor bug in `doreact` which allowed the molecule superstructure indices to become illegal if not enough molecules were allocated.
- Fixed a minor bug in `cmdreact1` which did not check for errors from `doreact`.
- Added command `molpos`.
- Moved version number from a `printf` statement to a macro, in smoldyn.c file.
- Added command `listmols2`, from a file sent to me by Karen Lipkow.
- Fixed a minor bug in `checkparams` that printed warnings for unused reactions.
- In `simulatetimestep` in smoldyn.c, the order of operations was `diffuse`, `checkwalls`, and then `assignmolecs`. The latter two were swapped, which should make wall checking more accurate when time steps are used that are so long that rms step lengths are a large fraction of box sizes. The new version is less accurate than before when the simulation accuracy is less than 10, but should be more accurate when it is 10.
- Replaced the `coinrand` call in `unireact`, which determines if a reaction occurred, with `coinrand30` to allow better accuracy with low probabilities. Also changed the relevant check in `checkparams`.
- Improved reactive volume test in `checkparams`.
- Increased `RANDTABLEMAX` from 2047 to 4095.
- Some modifications were made to random.h.
- Fixed a major bug in `rxnfree`, regarding the freeing of the `table` elements.

## Modifications for version 1.52 (released 10/24/03)

- Changed comments in rxnparam.h and rxnparam.c, but no changes in code.
- Changed `cmdsavesim` in smollib2.c to allow it to compile with gcc.
- Added another call to `assignmolecs` in `simulatetimestep` in smoldyn.c, after the call to `checkwalls`, to make sure that all molecules are assigned properly before checking reactions. This slows things down some, but should allow slightly longer time steps.
- To the opengl2.c file, the `KeyPush` function was modified so now pressing 'Q' sets the `Gl2PauseState` to 2, to indicate that a program should quit. A few modifications were also made in smoldyn.c function `TimerFunction` to make use of this.
- Corrected two significant bugs in the `checkwalls` function in smollib.c regarding absorbing walls. First, it didn't work properly for low side walls. Also, the probability equation was incorrect, which was noticed by Dan Gillespie.
- Fixed a minor bug in `cmdsavesim` in smollib2.c file, which caused an output line for `rate_internal` to be displayed for declared but unused reactions.
- Several commented out functions in `loadrxn` were removed because they were obsolete and have been replaced by `product_param`. They were: `p_gem`, `b_rel`, `b_abs`, `offset`, `fixed`, and `irrev`.
- A command superstructure was created, which moved several structure elements out of the simulation structure. No new functionality was created, but the code is cleaner now. New routines are `cmdssalloc` and `cmdssfree`. Updated routines are: `simalloc`, `simfree`, `loadsimul`, `setupstructs`, `cmdoutput` (including function declaration), `openoutputfiles` (including function declaration and ending state if an error occurs), `commandpop` (including function declaration), `checkcommand`, `endsimulate`, `savesim`, `main`, and all commands that save data to files.
- Renamed the "test files" folder to "test_files".

## Modifications for version 1.53 (released 2/9/04)

· Cleaned up commands a little more by writing routine `getfptr` in smollib2.c and calling it from commands that save data, rather than repeating the code each time.

· All routines that dealt with the command framework were moved to their own library, called SimCommands. This also involved a few function name and argument changes, affecting smoldyn.c, smollib.c, smollib.h, smollib2.c, and smollib2.h.

· Formatting was cleaned up for structure output routines.

· Swapped drawing of box and molecules, so box is on top. Also increased default box line width to 2 point.

· Computer now beeps when simulation is complete.

· Modified SimCommand library so that each invocation of a command is counted and also changed declaration for `docommand` in smollib2. This change was useful for improving the command `listmols2` so it can be run with several independent time counters. Also, wrote command `listmols3`.

· Wrote the new configuration file statement `boxsize`.

· Wrote the new commands `excludesphere` and `includeecoli`.

· Wrote the commands `overwrite` and `incrementfile`, which also involved some changes to the SimCommand library and required the new configuration file statement `output_file_number`.

· Added a new configuration file statement `frame_thickness`.

· When simulation is paused using OpenGL, the simulation time at which it was paused is now displayed to the text window.

## Modifications for version 1.54 (released 3/3/04)

· Swapped order of commands and OpenGL drawing so that commands are executed before displaying results. Also wrote section 3.2 of the documentation to discuss this ordering and other timing issues.

· Wrote documentation section 3.3 on surface effects on reaction rates and added the reactW set of test files.

## Modifications for version 1.55 (released 8/20/04)

· Improved graphics manipulations and added ability to save image as a TIFF file. This is not documented yet.

· Made a few tiny changes in random.c and string2.h and .c.

· The configuration file statement `max_cmd` is now obsolete because the command queue is automatically created and expanded as needed. Also, lots of changes were made to the library file SimCommand.c so that there are now two command queues: one is as before and uses floating point times for command execution and the other uses an integer counter for commands that are supposed to happen every, or every n'th, iteration.

· Added error strings to commands as well as the macro statement `SCMDCHECK`.

## Modifications for version 1.56 (released 1/14/05)

- Made lots of changes in opengl2.c.
- `#include` files for gl.h and glut.h now use brackets rather than quotes.
- Improved graphics significantly.
- Rewrote `TimerFunction` to clarify code.
- Added ability to save TIFF stacks which can be compiled into movies.
- Added `keypress` command.
- Added comments to the code.
- User and programmer parts of documentation were split to separate files.

## Modifications for version 1.57 (released 2/17/05)

- Added command `setrateint`.

## Modifications for version 1.58 (released 7/22/05)

- Fixed 2-D graphics so they a border is now shown again around the simulation volume.
- Added runtime commands `replacevolmol` and `volumesource`.
- Random number table for diffusion is now shuffled before use, which significantly reduces errors from an imperfect random number generator.
- Added position ranges to `mol` command.

## Modifications for version 1.59 (released 8/26/05)

- Random number seed is now stored and is displayed before a simulation starts.

## Modifications for version 1.60 (not released, but given to Karen 9/30/05)

- Fixed a small bug in `checkparams`.

## Modifications for version 1.70 (released 5/17/06)

- Added reflective, absorbing, and transparent surfaces for 1 to 3 dimensions with panel shapes that can be: rectangle, triangle, and sphere.
- Geometry.c and its header Geometry.h are new libraries that are used.
- Added background and frame color options.
- Reformatted and significantly updated part 2 of the Smoldyn documentation. Added surface descriptions to part 1 of documentation.
- Changed molecule sorting in `molsort` so that list compacting maintains list order.
- Wrote `reassignmolecs` to replace `assignmolecs`, which should increase efficiency and allow accurate surface treatment.
- Made it possible to load molecule names individually rather than all at once. New configuration file statements are `max_name` and `name`.
- Added pointers to the live molecule lists called `topl` (and renamed `top` to `topd`), which will differentiate old molecules from the new "reborn" ones. This is important for treating surfaces after reactions.

## Modifications for version 1.71 (released 12/8/06)

- Added `glutInit` call to `main` function in smoldyn.c.
- Changed OpenGL drawing slightly for surfaces, so now 3D surface colors are always the same on the front and back, but can also be semi-transparent, although with OpenGL errors.
- Added command `killmolinsphere`.
- Cleaned up simulation loading some, with minor modifications in `setupstructs`, `loadsimul`, and `setupboxes`, as well as writing of `setdiffusion`. This makes it so that molecule sorting only happens in `molsort`, and it took some unwanted code out of `loadsimul`.
- Added molecule serial numbers to the molecule structure and superstructure.
- Added some elements to command structures so that commands now have storage space.
- Added RnSort.c library to project, as well as some new functions in RnSort.c.
- Added command `meansqrdisp`.
- Completely rearranged order of functions in smollib.c and in documentation part II.
- Cleaned up surface code. Fixed rendering of 3-D spheres. Added support for cylinders and hemispheres.
- Added statements: `grid_thickness` and block comments with `/*` and `*/`.
- To `action_front` and similar statements, allowed "all" for molecule name.
- Tried to stop diffusing molecules from leaking across reflective surfaces.

## Modifications for version 1.72 (released 2/26/07)

- Finally got reflective surfaces to stop leaking diffusing molecules. This involved many changes in the surface code sections.
- Walls are no longer functional when any surfaces are defined, so new surfaces have to be defined to serve as system boundaries. Also, periodic surfaces are now possible.
- Changed all float data types to doubles throughout smollib.c, smollib2.c, Geometry.c, smoldyn.c, and their headers.
- Made it so that bimolecular reactions across surfaces can only happen with transparent surfaces.
- Two dimensional graphics now allow panning and zooming.
- Updated `savesim` command to accomodate surface changes.

## Modifications for version 1.73 (released 9/25/07)

- Trivial bug fixed in `loadsurface`, fixed minor bug regarding periodic surfaces.
- Fixed significant bug regarding 2D triangle surfaces.
- Improved `surfaceoutput` so that it only prints first 20 panels of each type for each surface.
- Added new commands: `settimestep`, `beep`, `echo`, `killmolprob`.
- Modified command `meansqrdisp` to read a dimension number. Also made it so it accounts for periodic boundaries.
- Added 'x' to possible command stepping options; then improved it 5/25/07.

- Added `wrap` element to molecules and implemented it.
- Lots of trivial changes so that Smoldyn would compile with gcc using "-Wall" flag without warnings.
- Smoldyn can now compile without OpenGL and/or without libtiff.
- Improvements to Makefiles and improvement of compiling advice.

## Modifications for version 1.74 (released 10/22/07)

- Coincident surfaces have defined behavior.
- New statement `boundaries` replaces `low_wall` and `high_wall` (but old ones still work).
- New command: `warnescapee`.
- Disk shaped panels are now possible.
- Surface jumps are now possible, which allow for holes in surfaces as well as for better periodic boundaries.
- Significant bug fix in reflections at circular and spherical surfaces.
- If 50 surfaces are encountered by a molecule in one time step, a warning is now printed.
- Cleaned up a lot of surface code.
- Replaced the surface statements `action_front`, `action_back`, and `action_both` with `action`; `color_front`, `color_back`, and `color_both` with `color`; and `polygon_front`, `polygon_back`, and `polygon_both` with `polygon`.

## Modifications for version 1.75 (released 11/6/07)

- Added surface-bound molecules. This has involved changes in the molecule superstructure, surface structures, panel structures, and other structures, changes in the graphics to display these states, new input statements (`surface_mol`, `epsilon`, `action`, `neighbor`, changes to `color`, `display_size`, `difc`, `surface rate`, `surface rate_internal`, reaction statements, and others), and a lot of new functions in the code. Nearly every function in smollib.c required at least some changes to account for this addition. Many additions have not been debugged yet.
- New library dependency: Sphere.c and Sphere.h.
- Rewrote the graphics code for drawing molecules and part of the code for drawing surfaces.
- Changed panel names, so now they are unique for each panel within a surface, not just each panel within a shape. Also, panel names are available from panels.
- Removed the `assignmolecs` function, which was superceded by `reassignmolecs`.
- Changed several character type structure elements to enums, including `PanelShape`, `PanelFace`, and `SrfAction`.
- Made it so the `read_file` statement can be used within surface or reaction statement blocks. This included adding a short Parse section and structure to the code.
- Renamed `test_files` folder to examples. Also, updated the configuration files so that they run with the current syntax.
- Improved error reporting, so now the problematic line of text is displayed.
- Cleaned up reaction loading by adding new functions that allocate reaction structure space for reactants and products.
- Starting adding a user manual section to the documentation.

## Modifications for version 1.76 (released 11/7/07)

- Split smollib.c and smollib.h source code files into smolload.c, smolrun.c, and their headers. Also, moved some functions from smoldyn.c into the libraries.
- Renamed some functions.
- Added and implemented more command-line flags.
- Put most global variables into sim, but killed off the event counters.

## Modifications for version 1.77 (released 11/18/07)

- Overall, few changes that affect users.
- Made it so surfaces outside of the simulation volume are checked, just like those within the volume. This fixes errors that can occur.
- Rewrote `checkwalls` so that it's more robust for molecules with long rms step lengths and so that its inputs mimic `checksurfaces`.
- Modified `line2nextbox` so that it accounts for boxes that are outside of the system volume.
- Made most files in smolrun.c so that they can operate on either complete molecule lists or only those that are in single boxes. This may help for parallelization.

## Modifications for version 1.78 (released 11/29/07)

- Minor changes with graphics.
- Changed the `overwrite` and `incrementfile` commands so that they no longer run a second command.
- Added and implemented the `CMDcode` enumerated list in SimCommand.c. Also in SimCommand.c, added an `iter` parameter to `scmdexecute` and aded the functions `scmdcmdtype` and `scmdnextcmdtime`; these may be useful for parallelization.
- Fixed a bug in surface actions.
- New functions: `readmolname`, `molsetdifc`, `molsetdifm`, `molsetdisplaysize`, `molsetcolor`. These clean up code that was elsewhere, allowed more "all" input parameters, and simplify the linking of Smoldyn to other applications. Also, the input formats of some statements (`difc`, `difm`, `surface_mol`, `display_size`, `color`, `permit`, `products`) and a lot of commands were changed to use `readmolname`, which improves them.
- Fixed bugs in lots of commands, which were created by the addition of surface-bound molecules in version 1.75.
- Added function `molcount`, which was used to simplify and improve several runtime commands.
- Rewrote `savesim` command, which involved new functions: `writesim`, `writewalls`, `writesurfaces`, `writereactions`, `writecommands`, and `writemols`.
- Added command timing option 'j'.
- Added `sname` to surface structure and changed `jumpp` in panel structure to point to a panel.
- Added reaction reversible parameter 'X'.
- Fixed a major bug in `findreverserxn` that caused it to fail.
- Added checking to `checkrxnparams` for multiple bimolecular reactions with same reactants and for defaults used for the reversible parameter.

- The `permit` element in reactions was changed from size `MSMAX` to size `MSMAX1`, including also implementation in the `bireact` functions and reaction set up functions.
- Changed `addrxnstostruct` parameter list to make it easier to use and to support an improved reaction input format.
- Added command `molcountinbox`.

## Modifications for version 1.79 (released 12/6/07)

- Fixed several bugs in `scmdexecute`. Commands no longer execute repeatedly after the simulation is finished.
- (Several of the following corrections were suggested by Kevin Neff; thanks!)
- Added newlines to the ends of header files to avoid compiler warnings.
- Fixed types of some stand-in functions for compiling without OpenGL.
- In opengl2.c, `WriteTIFF` function, changed `uint32` to `unsigned int`. Also, changed inclusion of "tiffio.h" to <tiffio.h>.
- Changed the Makefile so that it is easier to configure for no OpenGL and/or no Libtiff. Also, included linked library option (thanks to Upi) and an option for converting line terminating characters from Mac to Linux.
- Added basic compartments.
- Added commands `molcountincmpt` and `molcountonsurf`.
- Fixed a bug in the `molcount` function.
- The `savesim` command now works again, at least mostly.
- Replaced all `%lf` format specifiers in `printf` statements with `%g` and `%lf` in `scanf` to `%lg`.
- Changed the program license from "permission is granted for non-commercial use and modification of the code" to LGPL.

## Modifications for version 1.80 (released 12/22/07)

- Changed the exiting code some so 'Q' quits the program.
- Commands that don't work now print errors only once and aren't repeated.
- Modified `setrates`, `setproducts`, and `setupsurfaces` so that the time step can be modified during the simulation. Also modified command `settimestep` to work properly.
- Command `savesim` now does not save the output file name in the list of `output_files`, which means that the saved result can be run without overwriting it.
- Implemented `rxn->permit` element in `setrates` and also fixed a bug in `setrates`. A slight change in `setrates` is that `rate2` was not overwritten if it already had a positive value; now it is overwritten.
- Added and implemented `Simsetrandseed` function.
- Wrote part of `checksurfaceparams` function.
- Finished basic compartment implementation. `setupcomparts` was written, as well as new functions `boxrandpos` and `compartaddbox`.

· Added basic molecule porting for supporting MOOSE, with a new "port" surface action. Currently, this is more of a hack than a proper addition. New functions include `molgetexport` and `molputimport`, as well as `mollistnum`, `moldummyporter`, and `getnextmol`. Also, there is an `nlist` element in molecule superstructures, the number of live lists was expanded from 2 to 3, and there is a `list` element in molecules. `molsort` required some changes, as did various other functions.

## Modifications for version 1.81 (released 1/22/08)

· Changed developing environments from Macintosh Codewarrior to Macintosh gcc with XCode as an editor.

· Restructured code from smolload.c, smolrun.c, and smollib2.c libraries, to smolmolec.c, smolsurface.c, smolboxes.c, smolgraphics.c and others. All headers are now in smoldyn.h.

· Rearranged and tidied up all code.

· Rearranged `else if` order in `loadsim` function.

· Replaced the fixed number of live lists (2 up to version 1.79, 3 for version 1.80) with a variable number. Also, changed the molecule superstructure significantly to implement these multiple lists. Most molecule functions required changes, as did some reaction functions, and many others.

· Added molecule porting properly instead of as a hack, which involved new structures, new functions, and a new source code file.

· Added a lot of new molecule functions, as well as some surface and reaction functions.

· Added statements `gauss_table_size`, `molecule_lists`, and `mol_list`.

· Added and implemented `prob` and `tau` elements to the reaction structure.

· Fixed many minor bugs throughout the code.

· Cleaned up various minor issues in diagnostics output.

· Added simulation event counters to `sim`, which involved minor changes in several top-level functions including `checksurfaces`, `checkwalls`, `zeroreact`, `unireact`, and `bireact`.

· Added and implemented event counters to the simulation structure.

· Lots of work went into part I of the documentation, along with many new and improved example files.

## Modifications for version 1.82 (released 2/28/08)

· Changed `readmolname` so now a name without a state implies `MSsoln` rather than `MSall`, as it was before.

· Fixed `bireact` and `unireact` so that now molecule identities are checked before reactions are performed to make sure that they didn't just get killed off in a prior reaction.

· Reactions were overhauled with all new configuration file commands, new data structures, a lot of new functions, new diagnostics output, changes in every function in smolreact.c, improved example configuration files, and updated text in the user manual. A bug was fixed in which reaction permissions did not always correspond to those that were entered. Another bug was fixed in which reaction products frequently escaped the simulation volume.

## Modifications for version 1.83 (released 3/14/08)

- Fixed minor bug with allosteric reactions.
- Changed order 1 reaction probabilities so that multiple reaction pathways now simulate with the correct rates (example file is unireactn.txt).
- Added command `setrandseed`.
- Actual surface action rates are now displayed.
- Verified that surface sticking rates are correct.
- Fixed a minor bug in `line2nextbox` function.
- Added drift to molecular motions.

## Modifications for version 1.84 (released 4/11/08)

- Fixed a very minor bug in `line2nextbox` function.
- Added `-o` command line option
- Changed command `meansqrdisp` so that it now outputs $\langle r^4 \rangle$ as well as $\langle r^2 \rangle$.
- Rewrote command for changing the time step, which also included some new set time step functions and changes in a few setup functions.
- Added an enumeration for Smoldyn structures (enum SmolStruct), as well as text input and output functions to support them.
- Made a minor change to `reassignmolecs`, so the function doesn't bother running if there is only 1 box.
- Added commands `killmoloutsidesystem`, `diagnostics`, `setgraphics`, and `setgraphic_iter`.
- Fixed a bug in `fixpt2panel` regarding cylinders with their fronts inside.
- Cross-compiled Smoldyn for Windows using mingw compiler.
- Added order 0 and order 1 compartment reactions.

## Modifications for version 1.85 (released 6/3/08)

- Changed command execution timing for 'x' type commands so that they do not execute any more often than the simulation time step.
- Added command `molcountspace`.
- Fixed some minor bugs in `srfcalcrate`.
- Replaced the use of the system-supplied random number generator with the SIMD-oriented Fast Mersenne Twister (SFMT).
- Fixed a bug in which molecules leaked out of the corners of sticky boxes.
- Fixed a bug that was created in version 1.84 in which reversible reactions led to leaky surfaces.
- Fixed a trivial bug in which the wall list wasn't freed upon program termination.
- Fixed minor bugs in circle and arc drawing code.

- Ran Valgrind on Smoldyn with several configuration files and found no memory errors in the Smoldyn code (although several are reported for OpenGL).
- Fixed a bug in `findreverserxn` for reactions with 3 or more products.
- Fixed a bug in `rxnsetproduct` regarding type `RPpgemmax` parameters.
- Cleaned up set up functions some, with new documentation.
- Fixed a bug in zeroth order compartment reactions.
- Parser code was separated from molecules source file, and smoldyn.h header file, and put into parse.h and parse.c. Also, this code got a lot of improvements, including support for macro substitution. New statements include `define`, `define_global`, `undefine`, `ifdefine`, `ifundefine`, `else`, and `endif`.
- Renamed allosteric reactions to conformational spread reactions. This included changes in smoldyn.h, smolreact.c, and the documentation.

## Modifications for version 1.86 (released 11/17/08)

- Small progress on implementing accurate adsorption algorithms.
- Added display size checking for molecules.
- Fixed a bug that arose if both `max_names` and `names` were used.
- Fixed a bug in `molcountspace` command.
- Fixed a bug in `define` statements
- Changed makefiles to not assume availability of SSE2 processors
- Changed queue.c source file so that the voidcomp.c source file is no longer needed; this should enable compiling on 64-bit computers without problems.
- Fixed a bug in which molecules that were created only through commands did not interact with surfaces.

## Modifications for version 1.87 (released 12/7/08)

- Fixed a minor bug in line parsing.
- Vastly improved the wrl2smol utility program.

## Modifications for version 1.88 (released 1/16/09)

- Added accurate adsorption, desorption, and partial transmission algorithms.
- Added commands `fixmolcount` and `fixmolcountonsurf`, `molcountheader`.
- Fixed minor bug regarding 256+ character config. files lines.
- Smoldyn now recognizes Macintosh format config. files.
- Added SmolCrowd utility program to the distribution.
- Added areas to surface outputs.
- Changed config. file molecule names from “names” to “species”.
- Simplified config. file for `start_reaction`, `start_surface`, `start_compartment`, and `start_port`.
- Added command timing options A, B, &, I, E, and N. Also, fixed a minor bug in integer command queue timing (these were changed from ints to long long ints).
- Added compartment definitions that logically combine other compartments.

## Modifications for version 1.89 (released 2/11/09)

- Wrote documentation for "simulation settings" and started documentation for "network generation".
- Added some code for libMoleculizer linking. This is very preliminary.
- Added surface-specific reactions, and also made compartment-specific reactions work for bimolecular reactions.
- Completed support for adsorption, desorption, and partial transmission. This required many bug fixes and lots of work on SurfaceParam.c. All of these functions were tested and seem to work well. Also, reasonably thorough quantitative tests were run for all algorithms and all appear excellent.
- Lots of work on code modularization. This involved some data structure modifications (moved the `nspecies`, `maxspecies`, and `spnames` elements from the `simstruct` to the molecule superstructure), lots of cleaning up in smolmolec.c, and a lot of changes in `loadsim`. One result is that many config. file statements that could only be entered once before can now be entered multiple times, and each time overwrites the last one. This work is needed for a command line user interface and for Moleculizer linking.
- Progress towards making the setup functions able to run more than once, to allow for mid-simulation updates.
- Removed math2.h dependency of rxnparam.c file, and also improved documentation for it.
- Removed the `confspread` flag from the reaction structure since it was redundant with `rparamt` being equal to `RPconfspread`.
- Modified the `molcountspace` command slightly so that its spatial domains no longer include the endpoints; i.e. to (low,high), from [low,high].
- Changed surface molecule lists so that surface-bound reaction products are now checked for their positions after reactions and surface-bound actions (e.g. desorption) is handled better and more efficiently.
- Added "bounce" reaction product placement option. Molecules can now have excluded volume.

## Modifications for version 2.00 (released 2/17/09)

- Fixed a trivial bug so that the `time_start` statement now sets both the start time and the current time.
- Updated the version number to 2.00 to reflect the fact that all aspects of simulations with surfaces are now complete, as are a tremendous number of other improvements.

## Modifications for version 2.01 (released 3/3/09)

- Fixed a bug with compartment volume calculations, and improved compartment setup. Renamed `compartaddbox` to `compartupdatebox`, and largely rewrote it.
- Fixed a bug in SurfaceParam.c regarding rate calculation for reversible transmission.
- Added command `setsurfcoeff`, which Zsuzsanna Sukosd wrote.
- Separated `loadsim`, `loadsurface`, `loadcompart`, and `loadport` into two functions each, where the new functions are `simreadstring`, `surfreadstring`, `compartreadstring`, and `portreadstring`. Also added new statements to make the use of statement blocks optional within configuration files.
- Added command `set`.

- Made it possible for panel neighbors to be on other surfaces. As part of this, wrote function `surfchangeneighbors`, along with a few other functions.

- Substantially revised and improved surface diffusion. Checked all 2-D and 3-D panel shapes for basic functionality.

## Modifications for version 2.02 (released 5/5/09)

- Changed `molcount` function so that it now works for unsorted molecule lists as well as for sorted molecule lists. This will affect most molecule counting commands.

- Some code modularization so that most statements in `simreadstring` now work through functions rather than directly. More work is needed for `surfreadstring`, in particular. Also, decreased dependence of modules on each other.

- Added and implemented a `condition` element to each superstructure, which will allow for improved modularity. This will also allow changes during run-time, such as new species.

- Fixed bugs in which reactions between solution-phase and surface-bound molecules didn't always happen.

- Added command `molcountincmpt`.

- Added command `meansqrdisp2`.

## Modifications for version 2.03 (released 5/22/09)

- Changed build system from a simple makefile to the GNU Autoconf/Automake system. This works for many systems but not all.

- Added parallel operation using pthreads. This seems to work well, but doesn't actually lead to much speed up.

- Added support for libMoleculizer, which includes a lot of new code.

## Modifications for version 2.04 (released 6/27/09)

- Lots of improvements to the same changes that were made for version 2.03. Now, all of these features appear to be stable and the build system appears to work for most Mac or Linux systems. However, some documentation is still lacking.

- Added boundary absorption for effective unbounded diffusion.

- Improved SurfaceParam.c some to remove all dependencies and to make it run a little faster.

## Modifications for version 2.05 (released 7/23/09)

- Improved Smoldyn build system so that it now works for most Mac and Linux platforms.

- Some updates to the documentation, particularly in the compiling section.

- The Smoldyn release now includes the libmoleculizer documentation.

- Changed Smoldyn license from LGPL to GPL because of realization that large parts of the code could easily be used in proprietary software, which would hurt development of the Smoldyn project.

## Modifications for version 2.06 (released 11/6/09)

- Lots of code cleanup, including formatting changes, removal of unused variables, adding function declarations, etc.
- Subversion site cleanup, with removal of unimportant files.
- Added jump behaviors for peripheral surface-bound molecules.
- Improved conversions between strings and enumerated types for surfaces.
- Added a graphics time delay for simulations in pause mode or completed simulations.
- Added optional species conversion at surfaces.
- Fixed a bug in molecule existence checking.
- Tidied up execution time reporting.
- Fixed several minor bugs in parameter checking.
- Fixed a bug in which the maximum number of species was incremented twice for empty molecules.
- Got compilation for Windows to work again, using a hand-written makefile (no libmoleculizer support).
- New release shell script.

## Modifications for version 2.07 (released 11/17/09)

- Trivial updates to `surface_mol` and `compartment_mol` (to support 0 molecules)
- Fixed a bug in `molexpandlist`.
- Fixed a bug in `setupmols`, regarding `fixedlist` and `diffusinglist`.
- Fixed a minor bug in `rxnsetproducts`.
- Fixed several bugs in `savesim` command and added `writemolecules` and `molpos2string`.
- Fixed a major bug in diffusion of surface-bound molecules between panels.
- Fixed a bug in Geometry.c, in finding closest point on ring.
- Fixed a bug in reaction product placement, which shouldn't affect function but should speed code up.
- Fixed a bug in execution time reporting, and added time output to diagnostics.
- Fixed a minor bug with `neighbor_dist` statement.

## Modifications for version 2.08 (unofficially released 11/20/09)

- Fixed a bug for reactions between 2 surface-bound molecules, where the destination panel was sometimes wrong.
- Cleaned up release files some.

## Modifications for version 2.09 (released 1/6/10)

· Nathan fixed "make dist" build function.

· Fixed a small bug in wrl2smol, in which it didn't print out the input file name.

· Fixed a bug in `unbounded_emitter` surfaces which caused Smoldyn crashes.

· Fixed a minor bug in `surfstring2dm` for "polygon none" statement.

· Added command `molcountincmpt2`.

· Added statement and data structure portions for surface shininess, surface edge stippling, and graphics lights.

· Spun off the graphics stuff from `simstruct` and put into new graphics superstructure.

· Substantial work on graphics, including new `opengl_better` graphics drawing option.

· Added function `readsurfacename`.

· Added command `movesurfacemol`. It had a bug during initial release, which was fixed for re-release (1/12/10).

· Fixed two minor bugs in opengl2.c that only applied to compiling without opengl support. Re-released 1/20/10.

## Modifications for version 2.10 (released 3/24/10)

· Lots of work on libmoleculizer. Mostly rewrote smolmoleculizer.c file, and redid most of the Smoldyn moleculizer data structures. More work is needed though.

· Fixed a few minor bugs.

· Fixed a bug in which cylinders weren't always put into boxes.

· Moved text from the `INSTALL_NOTES` file into the manual and killed the file.

## Modifications for version 2.11 (released 5/4/10)

· Nathan added `-lglut` flag to standard configure for building.

## Modifications for version 2.12 (released 6/10/10)

· Changed panel input so that a panel can be input multiple times and the new data will overwrite the old.

· Added commands `replacecmptmol` and `molcountincmpts`.

· Changed `molchangeident` to allow it to kill molecules.

· Changed `readmolname` to allow it to read empty molecules, and also updated the code where this function was called. Now, many commands allow "empty" as a molecule species.

## Modifications for version 2.13 (released 7/15/10)

- Fixed reaction output slightly, so that activation-limited reaction rate is displayed and binding radius is not displayed for order 1 reactions.
- Various bug fixes in smolmoleculizer.c.
- Changed all `molcount` commands to be able to handle `nspecies` changing during the simulation.
- Improved `molsetexist` function, and set existence for lmzr species.
- Tidied libmoleculizer-dependent commands `speciesstreamcountheader` and `speciesstreamcount`.
- Substantial libmoleculizer work. Nathan made generated species have complex-form names rather than mangled names and he improved Python error reporting. I improved diffusion coefficients for generated species, including several new Smoldyn statements. More work is needed though.

## Modifications for version 2.14 (released 7/18/10)

- Fixed `#ifdef` portion of queue.h so that it will compile on Windows correctly.
- Added default product placement parameter for continuation reactions.

## Modifications for version 2.15 (released 7/20/10)

- Fixed bug that I just introduced for version 2.14 with default product placement parameter for continuation reactions.
- Fixed bug that didn't allow libmoleculizer to run.

## Modifications for version 2.16 (released 9/24/10)

- Added statement `expand_network`.
- Lots of work on libmoleculizer documenting and examples.
- Fixed yet another bug in `rxnsetrate` regarding automatic setting of product placement.
- Fixed a bug in which desorbed molecules could escape being checked for surface crossings.

## Modifications for version 2.17 (released 11/19/10)

- Added compile flag `-lGLU` to configure.ac file, which seems to be necessary for Ubuntu systems.
- In SurfaceParam.c, `surfaceprob` function, `SPArevTrans` algorithm section, replaced `exp(2*c1*c1)* erfcdD(SQRT2*c1)` with `experfcD(SQRT2*c1)` to expand the input domain.

## Modifications for version 2.18 (released 1/6/11)

- Fixed a small bug in `molchangeident` function. Before, panel data was retained for molecules that desorbed to `MSbsoln`, whereas it should not have been.

## Modifications for version 2.19 (released 2/11/11)

- Converted programmer's documentation from Word to LaTeX.
- Changed build configuring so that Libmoleculizer is now disabled by default, but can be enabled using the flag `--enable-libmoleculizer`. Before, it was enabled by default, but could be disabled.
- Cleaned up and documented pthreads code that Nathan wrote in the smoldyn.h, smolsim.c, and smolsurf.c files. More work is still needed for other files.
- Cleaned up `simulatetimestep` function, for slight efficiency improvements and better code.
- Renamed unthreaded functions from Nathan's names, which included a "_unitary" portion of the name, back to what I had before.
- Moved code for putting surface-bound molecules back on their surfaces from `checksurfaces` to `diffuse`.
- Enabled surface collisions for surface-bound molecules. This involved adding the surface action data structure, redesigning surface rate inputs and outputs, and redesigning surface action and rate indices. It also involved lots of new functions in smolsurface.c.
- Wrote documentation section on the simulation algorithm sequence.
- Improved surface and surface superstructure allocation so that they can be called more than once and the `max_surfaces` statement is no longer required.
- Modularized code in smolsurface.c, so that all statements recognized by `srfreadstring` now call functions rather than assigning directly to data structures.
- Swapped positions of `PFnone` and `PFboth` in the enumeration definition (from 3 and 2, respectively, to 2 and 3, respectively). I don't think this will cause problems, but I'm not certain.
- Added color words for all color inputs.
- Edited SimCommand.c to enable dynamic allocation for output files.
- Added and enabled `append_file` statement.
- Added `ifincmpt` and `killmolincmpt` commands.
- Added "–define" option for command line.

## Modifications for version 2.20 (released 3/4/11)

- Fixed bug which causes crashes if species change at surface was to "empty".
- Fixed a major bug, which caused Smoldyn to crash if it was compiled without threading support (this included the Windows version).
- Threading is now disabled by default.
- Fixed a few define statement issues. Trailing whitespace after replacement text is no longer part of the replacement. Multiple replacements no longer occur in the same piece of text (e.g. the AB of ABC is replaced with DE, making DEC, and then the EC is replaced with FG to yield DFG). Finally, long keys now take priority over short keys when both are able to match the same text.
- Added display_define statement.

## Modifications for version 2.21 (released 3/11/11)

- Fixed bugs in Geometry.c for nearest triangle column and nearest triangle points.
- Trivial change to opengl2.c, so that it now displays the file name as the window name.
- Fixed rare bug in which line2nextbox couldn't find a next box due to roundoff error.
- Enabled diffusion for surface-bound molecules from spherical or similar panels to neighboring panels that happens when a collision occurs (new code in dosurfinteract).
- Increased the number of panel neighbors that can be entered on one line from 32 to 128.
- Improved but did not fix problem in `checksurfaces` where a molecule interacts with surface A and when it is placed on the correct side of surface A it is inadvertently placed on the wrong side of surface B due to round-off error.
- Improved SmolCrowd user interface.

## Modifications for version 2.22 (released 3/22/11)

- Fixed a trivial mistake in opengl2.c that I created in version 2.21 that arose when compiling without opengl.
- Worked on shared library compilation and hand-coded makefiles. Added libsmoldyn.c and libsmoldyn.h to the Smoldyn source and added the libsmoldyn directory to the Subversion site and distribution.

## Modifications for version 2.23 (released 6/24/11)

- Wrote most of libsmoldyn.h and libsmoldyn.c.
- Improved modularization and dynamic memory allocation for compartments.
- Improved modularization and dynamic memory allocation for ports.
- Added commands shufflemollist and shufflereactions.
- Fixed some minor bugs regarding jump panel display and error checking.
- Improved robustness and conciseness of molecule parameter display.
- Added display of time and molecule counts to graphics window.
- Cleaned up conditional compiling switching in opengl2.c and smolgraphics.c, so that each function only has one opening line, which is the same for all configure options.
- Added default product placement type for bimolecular reactions with two products to RPpgemmaxw.
- Added commands `ifflag`, `ifprob`, and `setflag`.
- Removed the need for max_species and max_mol statements by implementing automatic memory allocation for both.
- Added a `simupdate` call from `simulatetimestep`, which will take care of data structure changes that commands cause.
- Fixed a minor bug in parse.c, Parse_ReadLine, for define statements that have no replacement text.

- Renamed, edited, and wrote several functions so that each module now has a function for updates, and this updating function now calls separate functions for updating at the lists level and at the parameters level. `setupmols` became `molsupdate`, `molcalcparams` became `molsupdateparams`, `setupboxes` became `boxesupdate`, `setupcomparts` became `compartsupdate`, `setuprxns` became `rxnsupdate`, `rxnsetmollist` became `rxnsupdatelists`, `rxnsettimestep` became `rxnsupdateparams`, `setupsurfaces` became `surfupdate`, `surfsettimestep` became `surfupdateparams`, and `setupports` became `portsupdate`.
- Checked entire code to make sure that data structure condition elements are modified correctly upon system changes. With these changes, all system changes, performed either before or during simulations, should trigger the necessary updates and thus take care of themselves.
- Edited Parse_ReadLine so that it now checks for lines that are too long and returns errors. Comment lines are allowed to be any length.
- Improved rxnsetproduct so that it now sets unbindrad and prdpos to 0, rather than assuming that they already were 0. Also, it copes better with conformational spread and other reaction types.
- Added checks for order 0 reactions with surface-bound products and no defined surface. Bug reported by Christine Hoyer.
- Added command listmolscmpt.
- Fixed a bug in which reactions between surface-bound molecules and solution-phase molecules could result in products that went to the wrong side of the surface. This involved changes to the panel assignment of the reaction position, and the reaction position, in doreact.
- Added a margin to each panel, so that diffusing surface-bound molecules that need to get moved back to panels that they diffused off of (or onto neighboring panels) no longer get moved to the exact panel edge, but to distance margin inside of the panel edge. Adding this involved adding a margin element to the surface superstructure, adding a margin statement, adding the `surfsetmargin` function, and adding margin parameters to `movept2panel` and `movemol2closepanel`.

## Modifications for version 2.24 (released 7/27/11)

- Modularized graphics input, so all assignments are now made in smolgraphics functions rather than directly in `simreadstring`.
- Added basic graphics support to libsmoldyn.
- Edited `unireact` so that it now tests for reaction probability first, and also it tests for compartments and surfaces independently. This version should be slightly faster than prior ones.
- Libsmoldyn now compiles as both a static and a dynamic library with the GNU autotools. This is mostly from additions to Makefile.am in the source/Smoldyn directory.
- Lots of edits to configure.ac. This included some cleaning up, adding –enable-libsmoldyn option, and removing search for pre-installed Libmoleculizer. I also added an option for –enable-swig, although that will probably need modification soon.
- Added and implemented condition element in graphics superstructure.
- Moved most of the contents of smolsimulategl function into new graphics updating functions.
- Moved top level OpenGL functions out of smoldyn.c and into smolgraphics.c.
- Removed function declarations from smoldyn.h and put them in smoldynfuncs.h. This hides them from Libsmoldyn users.
- Removed library dependencies from smoldyn.h.

- sim-¿cmds is now a void* instead of a cmdssptr. This requires type casting in the code, but was required to remove a Simcommand.h dependency from smoldyn.h.

- Removed GLfloat data types from the graphics superstructure to remove an OpenGL dependency from smoldyn.h.

- Fixed all OpenGL data types in smolgraphics.c so that they will work even if, for example, GLdouble differs from double. I have not fixed data types in opengl2.c yet.

- Status note: both Smoldyn and Libsmoldyn compile well with AutoTools. For Mac, this includes all configure options, including with Libmoleculizer enabled. However, Libsmoldyn doesn't seem to actually work when it calls Libmoleculizer. With MinGW, both Smoldyn and Libsmoldyn seem to compile well and install to the windows directory, although not with Libmoleculizer. The Libsmoldyn build with MinGW is suffixed with .a; I don't know if it's supposed to have a .dll, nor how to create such a thing.

- Added molcountspecies and mollistsize commands.

- Wrote release.sh shell script for releasing, including building pre-compiled versions for the Mac and windows directories. Also wrote install.sh shell script for Mac version.

- Added simversionnumber and smolGetVersion functions.

- Got SWIG working for Python, although more work is needed in AutoTools stuff for SWIG code.

## Modifications for version 2.25 (released 9/26/11)

- In configure.ac file, line 278, swapped sequence of a couple of lines. For version 2.24, the 3 lines starting with `"# This is probably redundant"` were first and then the 2 lines starting with `OPENGL_CFLAGS`. I swapped this sequence. See e-mail 9/13 with Pascal Bochet regarding compiling on Ubuntu. Apparently, this change did not fix his problem, but instead he added "LIBS="-lglut -lGLU"" to his ./configure line and that worked.

- Added functions `surfexpandmaxspecies` and `rxnexpandmaxspecies`, and also edited `molenablemols` and `rxnssalloc`. This should fix bugs that arose when users added species without allocating memory using max_species.

## Modifications for version 2.26 (released 3/2/12)

- Added some support for species name entry using enhanced wildcards. This included new matching functions in string2.c, a logic expansion function in string2.c, the function `molwildcardname`, additions to many parameter setting functions, and additions to `molcount`. Support is fairly limited so far, and needs to be added to lots of commands and reactions.

- Added `simfuncfree` function.

- Libmoleculizer is close to working fully, including returning correct reaction rates. Membrane-bound states still pose problems. Also, checking has still been minimal.

- Fixed bug that made it impossible for excluded volume reactions to occur across periodic boundaries. In the process, also slightly improved reaction location determination for reactions that occur across periodic boundaries.

- Fixed a minor bug with define statement in which recursive defines sometimes got overlooked.

## Modifications for version 2.27 (released 7/26/12)

· Merging in VCell changes.

· VCell people are now accessing Smoldyn code from hedgehog.fhcrc.org server. They used my user name on 4/10/12 but should be using their own user names from 4/11/12 onwards.

· Changed Smoldyn to Cmake. This conversion still needs substantial work. It needs Libmoleculizer, it needs to compile SmolCrowd and wrl2smol, it needs to work with MinGW, and it needs to create statically linked binaries that will work elsewhere.

· Changed all for loops over enumerated types to include explicit type conversions.

· Major work on error handling. Moved CHECK and CHECKS from individual files to smoldynfuncs.h. Changed their definitions some and added CHECKMEM and CHECKBUG macros. The basic error handling design is changed. Now, functions typically catch errors using one of the CHECK macros. This writes an error string to the global variable ErrorString and sends control to the local failure label. At this point, functions clean up as needed and call simLog to report the error. simLog may throw an exception to higher up or may return. On the latter case, the function with the error passes an error code up to its calling function. The calling function usually catches this error with CHECK but does not report it with simLog because it has already been reported. As a result of this change, functions no longer pass strings with error messages, but instead messages always transmit via the global ErrorString function. The exception is for library functions, such as Parse.c.

· Replaced all printf functions in the core code with calls to simLog.

· Added triangle area function to geometry.c and replaced Ye Li's function with a more numerically stable one.

· Improved initial window placement in opengl2.c (Jim Schaff's change).

· Changed char* to const char* for C++ compatibility in: boxsetsize and lots of other functions.

· Added a `strict` parameter to the `panelside` function to fix bugs that arose from coincident panels that face in opposite directions.

· Changed molecular desorption from surfaces from a simple change in molecule state and position, to a killing of the original molecule and a creation of a new molecule with the correct new parameters. This causes the desorbed molecule to be checked for further surface crossings with the other reborn molecules, which prevents it from leaking out of the system in case it desorbs across a surface. Also, changed surface checking for reborn molecules so that all reborn molecules are checked and not just those that can undergo reactions.

· When existing molecules are replaced by new ones, which occurs now in desorption and also in bounce reactions and conformational spread reactions, they now keep their serial numbers and their posoffset vectors.

· Added support for molecular drift that is relative to the local panel orientation. This included the `surfdrift` part of the molecule superstructure and all of its setting, allocating, freeing, and functioning.

· For bounce type reactions, added a default behavior where the unbinding radius isn't fixed but is the binding radius plus the amount of overlap between the two molecules, when they were collided.

· Added a precision to the numerical output for commands.

· SCMDCHECK was changed to accept a format and message string, rather than just a static message.

- VCell group added lots of stuff to compartments. In particular, it can now use volume sampling to determine what fraction of each box is in which compartment, rather than testing random points. This is vastly faster. compartmentIdentifierpair name is compartment name and pixel is the compartment id. As part of this, VolumeSamples is struct in smoldyn.h. num[3] is size of grid on x,y,z. size[3] is the width of each volume element on each dimension. origin[3] is the grid origin. volsamples is array with each unsigned char equal to the compartment ID for the center point of that volume element. ncmptidpair is number of possible compartments (maximum ID number + 1). This functionality is not available in stand-alone Smoldyn. getcompartmentid is new function. posincompartment is rewritten. Lots of new stuff is on top. Uses volumesamples method. compartupdatebox is rewritten, and now called compartupdatebox_volumesample. VCell group added zlib dependency to unpack volume samples data. Also added fromHex function and loadHighResVolumeSample function. They made lots of changes in compartsupdateparams and added compartsupdateparams_volumesample.
- Added some error catching stuff to main.
- Added a separate VCell main
- Added a little stuff to sim struct.
- In retrospect, this release was okay, but the build system was far from adequate. It did not cross-compile for Windows and the compiled version did not work on Mac OS 10.5.

## Modifications for version 2.28 (released 8/28/12)

- Fixed a bug in `morebireact`, in smolreact.c, where molecule positions were moved to account for periodic boundary wrapping, but the `posoffset` element was not updated.
- Added `listmols4` command.
- Improved the build system. Now, zlib is not a standard dependency.
- product_placement can now be entered multiple times.
- Commands no longer output using fprintf, but now use scmdfprintf. This enables output using user-chosen precision. It will also be useful for output from simulations to libsmoldyn, but that hasn't been added yet. Because of this change, there is a whitespace change in essentially all text output files. As a result, the regression tests show that all files differ from version 2.27 to version 2.28. However, I went through the files and verified that there are no other changes.

## Modifications for version 2.29 (released 4/10/13)

- Fixed a minor bug in `molismobile`, in smolmol.c, which caused bugs for some species with surface drifts.
- Surrounded most VCell code with conditional compiling statements, so it's no longer part of standard Smoldyn builds. The reason was primarily so Libsmoldyn could be C compatible.
- Various minor Libsmoldyn fixes (smolAddReaction was missing a check on product states, changed default throwing threshold and debug mode behaviors, and added call to checksimparams to smolDisplaySim).
- Improved CMakeLists.txt file for Libsmoldyn compiling.
- Fixed CMakeLists.txt file so that it now installs Python libraries for Libmoleculizer.
- Added several new colors to the color words that can be used, in smolgraphics.c.
- Added command called executiontime.

· Renamed rand_seed statement to random_seed, but left it backward compatible.

· Added extern "C" stuff to libsmoldyn.h.

## Modifications for version 2.30 (released 8/21/13)

· Fixed a bug in `Geo_Cyl2Rect`, in the Geometry library. It was causing leaking surfaces.

· Added function `checksurfaces1mol`, for checking surfaces after a molecule hits a port. This also required minor changes in `molchangeident`, `molkill`, and `surfinteract`.

· Fixed a minor bug in `surfaddpanel` that didn't allow panels to be redefined.

· Changed the termination text because some users didn't understand how to quit properly.

· Added `meansqrdisp3` command.

· Added `ifchange` command.

· Added `residencetime` command.

· Changed license from GPL to LGPL to promote Smoldyn's use in other software. In particular, VCell can't use Smoldyn under the GPL license. Also, I think that NSF and other funding agencies prefer LGPL.

## Modifications for version 2.31 (released 9/9/13)

· Added command called setreactionratemolcount.

· Added a line to rxnsetproduct that enables product setting even if the reverse reaction has rate 0.

· Modified command meansqrdisp3 so that it weights diffusion coefficients based on molecule lifetimes and also so it works with all states of a species.

## Modifications for version 2.32 (released 8/29/14)

· Overhaul of wildcard support. This included work on the wildcard match and substitute functions in string2.c, addition of pattern data structure components in the sim-¿mols superstructure, addition of some pattern handling functions including `molstring2pattern`, `molpatternindex`, `molstring2index1`, and memory allocation functions for patterns and pattern lists, a rewrite of the `molcount` function, and implementation of wildcards in `simreadstring`. This task is not complete yet; see below in the wish/ to do list.

· Added lattice support by merging nsv branch into trunk. The following functions are new and some of these still require documentation: smolcmd.c: cmdwriteVTK, cmdprintLattice, edits to cmdmolcount, cmdmolcountspace, cmdsavesim; libsmoldyn.c: smolAddLattice, smolGetLatticeIndex, smolGetLatticeIndexNT, smolGetLatticeName, smolAddLatticeMolecules, smolAddLatticePort, smolAddLatticeSpecies, smolAddLatticeReaction; smolport.c: portgetmols2, portputmols2; all of smollattice.c; smolsurface.c: checksurfaces1mol; smolgraphics.c: RenderMolecs.

· Documentation was partly updated but both doc1 and doc2 still require many updates.

· Substantial editing of CMakeLists.txt file and release scripts.

· Added BioNetGen to Smoldyn utility programs. They still need work, but they're incuded in the distribution now.

· Removed Libmoleculizer, pthreads code, and the last vestiges of the AutoTools build system. None of these things ever worked quite as intended, so they were removed to clean up the code.

· Improved "bounce" reactions so that they now enable simulations of cluster formation.

· Wrote and added SmoldynQuickGuide to the Smoldyn documentation collection.

· Fixed a minor bug that recorded periodic boundaries in boxes even when surfaces were defined. This led to surface leaking.

· Fixed a minor bug in which the "keypress" command would cause crashes if the simulation was run without OpenGL support.

### Modifications for version 2.33 (released 10/9/14)

· Fixed a minor but important bug in changemolident.

· Made input names UK English compatible.

· Fixed typos in the QuickGuide.

· Updated all example files, so they no longer use reaction blocks and they don't use static memory allocation with max_ statements.

· Fixed a bug in which desorbed molecules could cross surfaces without detection.

· Added cmdtrackmol function.

### Modifications for version 2.34 (released 1/8/15)

· Nearly complete BNG conversion code.

· Added lattice code to default (pre-compiled) distribution.

· Fixed several small bugs.

· Changed lattice code so that molecules from continuous space to lattice space only get placed into first subvolume. This leads to more accurate diffusion results.

### Modifications for version 2.35 (released 4/23/15)

· Fixed substantial bugs in `rxnsetrate` and `rxncalcrate`, in smolreact.c. The bug was that unimolecular reactions that had multiple reaction channels (e.g. A $\rightarrow$ B and A $\rightarrow$ C) computed the conditional probability (the probability of reaction given that the prior reactions did not happen) for the second and subsequent reactions incorrectly. This was only a problem with relatively high reaction probabilities.

### Modifications for version 2.36 (released 6/9/15)

· Fixed a bug in `doreact` in which bounce reactions did not work when both reactants had exactly the same locations.

· Changed the default graphic iteration value from 1 to 20.

· All output commands now flush the output buffer after the command is done.

## Modifications for version 2.37 (released 10/7/15)

· Fixed a bug in parse.c in which global definitions weren't being passed to upstream files.

· Added a check at the cmd statement to make sure that simulation times have been entered.

· Fixed a bug in which surface-bound molecules diffused onto new panels incorrectly. This required quite a lot of work including many new functions in Geometry.c (all of the exit functions and several new triangle functions), a new function in Sphere.c (`Sph_RotateVectWithNormals3D`), expansion of the panel `points` elements to include edge normals, and lots of work in smolsurface.c. This work included: a complete rewrite of `movemol2closepanel`, additions to `surfaddpanel`, `movept2panel`, `closestpanelpt`, and the new `lineexitpanel` and `paneledgenormal` functions. I think all bugs are fixed but if not, see debugging advice in `movemol2closepanel`. One result of this fix is that surface-bound molecules that diffuse onto a panel that has a jump action no longer jump when that happens. That feature was added for Hugo's work, but I'm removing it at this point because I think there are better ways of accomplishing the same thing.

· Added reaction_serialnum statement and implemented it. This involved a small amount of code, mostly in smolreact.c.

· Added reaction_log and reaction_log_off statements and implemented them. This mostly involved a fairly small amount of code in smolreact.c. Also, I started a new library file called List.c for this. This addition, and the reaction_serialnum addition, should make it much easier to do particle tracking simulations.

## Modifications for version 2.38 (released 10/22/15)

· Added line to cmake file to enable building for Macs with OS 10.5 or above.

· Changed boundaries statement to allow 'x', 'y' and 'z' inputs for dimension values. Same change to low_wall, high_wall, panel rectangle, and some commands.

· Added updategraphics command.

· Added quit_at_end statement.

· Improved product_placement so that it now checks product states and it allows the user the specify products with "product_$n$".

· Fixed minor bug in parse.c that required all lines to end with
n, which didn't work if there was an end of file within the line.

· Fixed minor bug in simParseError that didn't return an error if a file wasn't found.

· Changed sim-¿cmds from a void* to a cmdssptr. I have no idea why it was a void* before.

· Improved reaction logging output by changing from fprintf to scmdfprintf.

· Edited scmdfprintf to allow for `NULL cmds` input.

· Edited `closestsurfacept` to allow for a box input.

· Fixed bug in `doreact` which occurred when the reactants were bound to two different panels, and the reaction occurred in yet a third panel. Also, substantially cleaned up `doreact`, `RxnSetRevparam`, and `rxnsetproduct` to make code clearer, more consistent, and generally better, and to improve documentation. Now, confspread and bounce reactions can have any number of products. Also, bounce reactions now measure the vector from the reaction position rather than from the center between the two reactants (typically the same, but not for curved surfaces). I suspect that bounce reactions are now as good as they can be made, within the limitations of the basic algorithm (e.g. only one bounce per time step).

- Fixed `doreact` so that reaction products that are surface-bound get moved to the surface right away.
- Changed parse.c so that Smoldyn can read Mac and, hopefully, Windows files without them needing to be converted to Linux format first.
- Added `systemcenter` function to smolwalls.c.
- Added expandsystem command and `surftransformpanel`, as first steps towards having moving surfaces. They work ok (at least the parts that I tested), but the lack of panel distortion with anisotropic expansion means that molecules can escape across surfaces in some cases.
- Added fixmolcountrange, fixmolcountrangeonsurf, and fixmolcountrangeincmpt commands.
- Added intersurface reactions, including the reaction_intersurface statement, the `RxnSetIntersurfaceRules` function, and implementation in `morebireact` and `doreact`.
- Fixed a minor bug in smollattice.c which arose when surface panel definitions were changed in version 2.37.
- Fixed bugs in `doreact` that caused surface-bound molecules in bounce type reactions to jump between surface panels. This was caused by incorrect setting of the new panel value.

## Modifications for version 2.39 (released 1/15/16)

- Fixed a bug in `Geo_NearestTrianglePt2`, which led to incorrect answers for points that were outside of the triangle. This fixed a bug that was detected in S7_surfaces/surfacediffuse/simple3a.txt.
- Added `cmdmolcountspaceradial` function.
- Made several very minor modifications requested by VCell. Mostly, changed #if OPTION_VCELL to #ifdef OPTION_VCELL, but also changed use of NAN and INFINITY.
- Added a check to `surfupdatelists` so that the `SMLdiffuse` flag is not set if all molecule-surface actions are "no" or "transmit" and there are no other details of interest. In other words, this prevents surface checking if there are no surface actions to check for.
- Modified `reassignmolecs`, from smolboxes.c, to make it much faster. Before, if a molecule needed to move boxes, this searched through the current box's molecule list for the molecule, which it then removed. Now, the same routine is used if only reborn molecules are assigned but otherwise this clears all box molecule lists and assigns them from scratch. This is vastly faster if boxes have lots of molecules in them.
- Fixed minor bugs in `meansqrdisp`, `meansqrdisp2`, and `meansqrdisp3` which did not allow computations for all axes.

## Modifications for version 2.40 (released 3/22/16)

- This is somewhat of a beta release, since a lot of things have changed and not all are documented or fully tested.
- BioNetGen support now works.
- Improved install for Mac slightly and Windows substantially.
- Added support for wildcards and formulas in essentially all statements and commands.
- Added "csv" output format and implemented it.
- Added ^ operator to `strmatheval` function.

· Edited `molpatternindex` and added `moladdspeciesgroup` function. Also implemented this with the new `species_group` statement.

· Added species groups.

· Added wildcard reaction expansion, including several new parts to the data structures.

· Removed `readmolname` and `molwildcardname` functions.

· Added `molgeneratespecies` function.

· Modified `graphicsenablegraphics` trivially so that graphics are not enabled by default.

· Improved graphics so surfaces in 3D now allow combinations of polygon values.

· Added `molcountspacepolarangle` command.

· Fixed several bugs in surface-bound molecule diffusion.

· Started updating the user's manual, but there's much more to be done.

## Modifications for version 2.41 (released 4/8/16)

· Fixed a trivial bug that caused crashes.

· Added a "Getting Started" chapter to the User's Manual and reformatted the whole manual.

· Added molecule-surface interactions for new species to the BioNetGen code.

· Fixed a bug in `molkill` calls in commands which prevented molecules from being killed.

· Added an improved bounce reaction algorithm, for ballistic reflection.

· Improved the Windows installation script. It might not work yet, but it's probably closer.

## Modifications for version 2.42 (released 4/29/16)

· Added command for radial distribution function, also including a `boxscan` function.

· Fixed bugs in excluded volume reacions, so they now work with periodic boundaries and have been thoroughly validated.

· Modified all listmols commands so that they now output molecule serial numbers. This causes all of the regression tests to report a different behavior, but that's because of the expanded output, not different simulation results.

## Modifications for version 2.43 (released 5/18/16)

· Renamed Smoldyn_doc1 to SmoldynUsersManual, and split Smoldyn_doc2 to SmoldynCodeDoc and LibsmoldynManual.

· Fixed excessive include files for Libsmoldyn. Also fixed some bugs in Libsmoldyn. It now works for the test files.

· Added gaussiansource and molcountspace2d commands.

### Modifications for version 2.44 (released 6/27/16)

· Added reversible reaction definitions. In the process, moved reaction parsing to the new function `rxnparsereaction` and added function `molreversepattern`.

· Fixed release.sh slightly so that freeglut.dll is included in the windows release.

### Modifications for version 2.45 (released 7/15/16)

· Fixed bugs in equilmol and modulatemol where they returned the wrong probabilities. These were caused by the change to using molscan in version 2.40.

· Fixed minor bug in molreversepattern that causing compiling errors on Red Hat Linux.

· Renumbered User Manual chapters to re-align them with the example file directories.

· Fixed a bug in strmatheval that read `2^-3` as a binary - sign when it should be a unary - sign.

· Fixed a bug in nsv_print in which it could only print to a fixed size buffer. It now allocates the buffer as needed.

· Added some checks to the lattice code to ensure that lattices have an integer number of compartments on each axis, that ports are at compartment boundaries, and that there is at least one compartment on either side of the port.

### Modifications for version 2.46 (released 7/30/16)

· Added section S94_archive to the examples files. It currently has files for my 2016 Bioinformatics paper.

· Added `multiplicity` element to the reaction structure and implemented it. As part of this, added a "reaction_multiplicity" statement to the input format.

· Major overhaul of `molpatternindex`. As part of this, it now has 2 levels of updating.

· Overhaul of `RxnTestExist`.

· On-the-fly wildcard expansion is now truly on-the-fly so that it expands around individual species as they are populated rather than expanding one entire level at a time. Added several files to the regression tests that all agree with the BioNetGen expansion, making them validated models.

### Modifications for version 2.47 (released 8/30/16)

· Added a new rules section to the code. This includes a rules portion of the data structure, the smolrules.c file, and functions in that file. In the process, I removed rules from the reaction superstructure. I also replaced the RxnAddRule and RxnExpandRules functions with RuleAddRule and RuleExpandRules functions. This is a cleaner design and should accomodate diffusion and other rule types.

· Added rules for molecule properties. These include for diffusion coefficients, diffusion matrices, drifts, surface drifts, molecule lists, display sizes, colors, surface actions, surface rates, and internal surface rates.

· Added a reaction template to the rule structure so now reaction modification statements, such as product_placement, can write to this template. The relevent portions of the template are then copied over when the rule is expanded.

· Expanded `readrxnname` so that it now returns reactions with the given name, rules with the given name, or reactions that start with the given name. Also, implemented this so that reactions entered with wildcards, whether rules or not, can be modified using reaction modification statements.

· Added `molcountspecieslist` command.

· Made the text_display statement functional with rules.

· Fixed bugs in `strEnhWildcardMatchAndSub` so that it now works with empty pattern or destination strings and also does one to many, many to one, zero to many, many to zero, many to many, etc.

· Updated a lot of commands so that if they request a species name and it doesn't exist, but there are reaction rules, then this isn't an error. Instead, an implicit assumption is made that the species will exist later on.

· Fixed bugs in reaction logging.

· Fixed a compiling bug in smolmolec.c in which const char* was being converted to char* in a strchr function.

## Modifications for version 2.48 (released 11/16/16)

· Fixed a minor bug in `strmathsscanf`.

· Added several example files in archive directory for Bioinformatics and MMB papers.

· Fixed bug in `latticeaddmols`.

· Added requirement that 'dim' has to be entered before reactions because not doing this causes crashes. It's likely required before other things, but I didn't modify others.

## Modifications for version 2.49 (released 2/17/17)

· Fixed bugs in molcount and molcountspace commands that arose if lattice species didn't align with Smoldyn species.

· Made it an error to call the reaction_log statement multiple times with the same reaction and different filenames. Previously, this was allowed and new filenames overwrote the old ones, but this behavior was confusing because it seemed as though there should be multiple output files.

· Wrote SmolEmulate, including 2D emulation code. This is not integrated into Smoldyn yet and is not ready for general release.

· Added `cmdtranslatecmpt` and `cmddiffusecmpt` commands. In the process, wrote several new functions for surface, panel, and compartment moving. Also, surfaces now have a list of their membrane-bound molecules.

· Added function parsing to string2.c, so Smoldyn now recognizes sqrt, sin, cosh, etc.

## Modifications for version 2.50 (released 2/27/17)

· Added infrastructure for Smoldyn functions. This involved the command `cmdevaluate`, the functions `fnmolcount`, `molscanfn`, `loadsmolfunctions`, and others. Also renamed `molscan` to `molscancmd|`.

- Added `oldpoint` and `oldfront` elements to surface panels. Now, when a panel is moved, the old location (oldpos) is stored in these old elements and the new ones are updated. Then, molecule adjustment is determined using the correct combination of old and new locations. Molecules still cross surfaces incorrectly if they get squeezed between a moving surface and a static surface (or two moving surfaces), but this is unavoidable. Several functions now include an `oldpos` input to see if the function should use the old or new panel positions.
- Added `if` command.
- Added `molcountonsurf` function.
- Added a molecule`touch` element that records each time the molecules were touched. It lets functions not recompute if the touch value is the same as it was during a prior call.

## Modifications for version 2.51 (released 3/15/17)

- Fixed a bug in which surface-bound molecules could leak through another surface. The problem was that the surface-bound molecule code was designed with relatively large panels in mind, and is generally exact for those panels, when they are planar. However, small panels are more complicated. As a result, I added the `pnlx` element to the molecule structure to keep track of the prior panel. Also, I changed the reflection algorithm in `dosurfinteract` so now the back-up plan in the case that reflection does not place the molecule on the correct side of the reflecting panel is to put the molecule back where it started from. The prior back-up plan was to fix the molecule to the correct side of the reflecting surface, but this made the system extremely sensitive to any future molecule adjustments. I also added the `fixpt2panelnocross` function which fixes a point to a panel but doesn't cross any other panels in the process. The final result works well, but it is still conceivable to have molecule leaks in rare situations.

## Modifications for version 2.52 (released 5/16/17)

- Martin Robinson fixed a minor bug in smolcmd.c that had to do with the lattice code.

## Modifications for version 2.53 (released 5/25/17)

- Fixed minor bugs in smolcmd.c that had to do with the lattice code. Part of the version 2.49 modifications was that I "fixed bugs in molcount and molcountspace commands that arose if lattice species didn't align with Smoldyn species." However, I now think the code was correct originally (at version 2.48) and that these changes for version 2.49 were incorrect. Martin effectively reverted one of them in molcountspace for version 2.52. I just finished that reversion and also reverted the one in molcount. Now, that portion of the code is identical to how it was in 2.48 and it seems to run well. I still don't know what prompted me to make these changes in the first place.
- Documented some of the nsvc.cpp functions.
- In strloadmathfunctions, which is in string2.c, the code now assigns function pointers to variables before sending them off to strevalfunction. This provides the context that C++ compilers need to determine which address to choose for the functions because their names are overloaded.

## Modifications for version 2.54 (released 8/27/17)

- Added a line to `molpatternindex` that sets the `PDrule` element of the `index` variable if update is requested, `index` already existed, and the element wasn't already set. The only thing this does is to prevent a bug warning in the output.

· Fixed two small bugs in lineXpanel for sphere and cylinder crossing checking. These failed if the two points were the same because then the geometry function returned NaN and then that failed the test. The solution was to move to negative logic so that NaN would have the correct result. These failure arose because I changed to fixpt2panelnocross and that checks for lines crossing panels, often with the same beginning and ending points. Added surfacedrift2 and 3 to the regression files because these were the ones that failed.

## Modifications for version 2.55 (released 7/16/18)

· Added a 'time' option to the `random_seed` statement for setting the value to the current time.

· Added math functionality for command timing.

· Trivial improvement to wall output dimensions from numbers to letters.

· Wrote bindingradiusprob and numrxnrateprob functions in rxnparam.c.

· The MinGW build system that I've been using for years (/opt/local/bin/i386-mingw32-gcc) quit working because I did a Mac OS upgrade last summer which somehow killed it (the upgrade was after Smoldyn 2.54 was released). After lots of effort, I found a MinGW version that was precompiled for Mac (an older OS, but that doesn't seem to matter), which I installed. It seems to work well. It's at /usr/local/gcc-4.8.0-qt-4.8.4-for-mingw32/win32-gcc/bin/i586-mingw32-gcc. I updated the Toochain-mingw32 file for this change.

· The Windows install wasn't working because the BioNetGen directory was called bin rather than BioNetGen. I think that's fixed by now.

· Added several lines of code to the adsorb section of dosurfinteract so that surface-bound molecules that adsorb onto a new surface end up on the correct side of the original surface, as well as on the correct side of the new surface. In particular, molecules starting in the up or down states end up equally distributed above and below the original surface.

· Largely rewrote `molsetlistlookup` so that it now sets the table for both `MSsoln` and `MSbsoln`. This change also required expanding the listlookup element slightly in the memory alloction function to allow for the `MSbsoln` option. The listlookup values are always set to the same lists for both solution phase options. As another part of this change, `molismobile` now allows for a `MSbsoln` input.

## Modifications for version 2.56 (released 9/18/18)

· Fixed a bug that arose when the system was expanded and then some molecules were killed. System expansion triggered boxupdateparams which then removed molecules from boxes but didn't clear the mptr-¿box value and then boxremovemol crashed due to the molecule not being in the box. Fix involved allowing for molecule not in box in boxremovemol and not adding dead molecules to boxes in boxupdateparams. In the process, I discovered that molecule assignment may be done more often than necessary.

· Removed a call to molsort from cmdgaussiansource. I'm fairly certain it was unnecessary, and tests showed that it still worked without it.

· Lots of struggles with MinGW. The version at /usr/local/gcc-4.8.0-qt-4.8.4-for-mingw32/win32-gcc/bin/i586-mingw32-gcc didn't work. I got it with MacPorts but that didn't work either. Yet more versions also didn't work. I finally killed off all of them and tried fresh. Solved MinGW compiling but not with LibTiff, which remains to be solved. See the MinGW cross-compiling documentation section.

· The MinGW compiler is more strict and found several minor bugs. Fixed some of them, but there are still possible string overwrite and truncation issues that it complains about.

- Implemented most of lambda-rho algorithm. As part of this, I did a lot of work on the SmolEmulate utility, computed reaction rate lookup tables for non-unit reaction probabilities, debugged the numrxnrateprob and bindingradiusprob functions, and then updated all calls to numrxnrate and bindingradius to these probabilistic functions. These updated functions call the original functions in the usual case that the reaction probability is 1. The functions in rxnparam.c are generally good now, but still not quite perfect.
- An another part of lambda-rho, added the input file statement "reaction_chi" and implemented it. This addition seems to work reasonably well, but Smoldyn isn't always able to align the simulated and requested reaction rates yet. In particular, it has trouble with the pgem, pgemmax, and pgemmaxw reverse parameters.

### Modifications for version 2.57 (not released yet)

- Fixed minor bugs in cmdwriteVTK that caused compiling errors when compiling with OPTION_VTK turned on.

# Chapter 7

# The wish/ to do list

## 7.1 Bugs and issues to fix

- Surfaces now have lists of molecules adsorbed to them. The code would be faster if these lists were used as appropriate, such as for checking the actions of surface-bound molecules.
- Smoldyn shouldn't terminate if pattern or species name is over-length during on-the-fly expansion, but ignore it and issue a warning.
- Should expand length from STRCHAR to STRCHARLONG for patterns because they are for reactions too.
- The binding radius if time step is 0 output doesn't account for surface-bound species.
- The unbinding radius if time step is 0 output doesn't always work (it sometimes displays -2). This may have been fixed for version 2.13.
- If `time_now` is not start time, then all commands from start time to current time get run. They shouldn't.
- Reaction molecule states should be improved. At present, `rxnXsurface` doesn't allow, for example, `fsoln` molecules to reaction with back molecules, but this is over-ridden in `bireact`. It would be better to use the reaction `permit` array more effectively, and also to allow users to enter relative reactions, such as same side, opposite side, surface-bound, etc.
- Excluded volume reactions do not work with periodic boundaries. Fixed for version 2.26. They also don't work with the `meansqrdisp` commands.
- Need to finish writing `checksurfaceparams` function. In particular: check probabilities, if panels are wholly outside volume, and if jump panels source and destination panels are compatible.
- 3-D box cross testing, done in Geometry.c, isn't correct for all shapes (triangles and disks in particular, maybe others).
- Surface rendering still has some problems.
- If there are multiple reaction channels for a first order reaction and if the rate of one is set as a rate while the rate of a later one is set with the probability, and then the simulation time step is then changed by a command, then the reaction probability of the later one does not properly account for the changed conditional probability of the first occurring. The solution is to add a data structure element called `probreq`, for probability requested, and to then calculate the simulation probability from this, so that the original user-entered information is never overwritten.

## 7.2 Desired features

### Libsmoldyn

- Test libsmoldyn.
- Modularize commands.
- Libsmoldyn needs functions such as: smolGetMolCount, smolGetMolCountCmpt, smolGetMolCountPort, smolGetMolCountSurf, smolGetMolPositions, smolGetNumSpecies, etc. Also, matched functions smolGetMolecules and smolSetMolecules.
- Prefix all pointer inputs with `const` if the memory that they point to won't be changed.
- Finish swigging for Python, and then swig for R and Octave.

### Code acceleration

- There are lots of little tweaks that could be made to speed up the code. For example:
- Replace most if...else constructs in the core simulation code with switch
- Replace mol-¿via with a local vector.
- Check for system dimensionality from 3 to 2 to 1, rather than in ascending order.
- Pre-compute and store the panel margin data with the panels rather than re-computing frequently.
- Unroll loops when possible, often creating separate code for 1D, 2D, and 3D.
- Try to replace division with multiplication and get rid of square roots and trig. functions, as possible.

### Electrostatics

- Arnd Pralle wants Smoldyn to simulate electrostatic interactions. At first, I was thinking that this would be a big pain. But then I realized that it's actually just a very minor tweak of the bounce type reactions. Instead of the product distance being some fixed distance or the collision distance plus the overlap distance, the product distance should be a function of the overlap. For attractive interactions, if the overlap is small, then the molecules get sucked in slightly, and if the overlap is large then they get sucked in more. An issue to deal with is that there will likely be multiple reactions listed for the same pair of molecules, and the right ones need to be executed at the right times. For example, there might be electrostatic interaction at relatively long distances but also chemical reactions at short distances.

### BioNetGen

- Need to finish groups stuff. More testing would be good, too.

### Math

- Add functions that have parentheses, such as sin(x) and atan2(x,y). Also add a molcount function.

### Commands

- Commands need to be overhauled. They need to be modularized, they need to support wildcards, and they need to be able to export data to Libsmoldyn.
- To commands, add an argument to the functions for data export by observation commands. I'm thinking that this should be a data structure that includes a header, maxcolumns, ncolumns, maxrows, nrows, and a data table. Also create a command adding function that Libsmoldyn can use to add commands to the system, and also get data back from commands.

- Simple spatially separated macromolecular complexes. Karen wants to be able to model Tar-CheA dimers, plus their interactions with their neighbors. My idea is to add a command which creates a "child" CheA for each Tar molecule, which are the "parents", and this CheA would have a fixed offset position from the Tar. When this command is called later on, every child is moved to the fixed offset value, thus causing the child molecules to always track their parents. This motion would ignore surfaces.
- It should be possible to change single reaction rates during a simulation. It's possible at present to change the internal value for the reaction rate, but not more user-friendly values. I think this is now possible in version 2.23 but I haven't verified it yet.
- It would be nice if commands could communicate with each other. An idea for this is to establish a bulletin board within the command superstructure, on which commands could post and read memos. More generally, this could be expanded into an entire programming language if desired, although it would take some thought on how to do it in the best way.
- A new runtime command for more versatile text output. Rather than having a pile of specialized output commands, it would be nice to have something akin to a print statement, where any of a wide variety of simulation variables could be printed with a user-defined format.

## Distribution

- Sourceforge. The account has been set up but needs code, etc. Then, I should e-mail everyone who might be interested to invite them to join the mailing lists.
- The Calibayes project might be a good place to integrate Smoldyn into, to enable parameter fitting.
- Need to improve Windows distribution. Write a batch file for installing. Note that the download location appears to be %USERPROFILE%\Downloads\smoldyn-2.xx-windows.zip. It should be moved to %PROGRAMFILES%. Instructions: user needs to right-click the downloaded zip file, which will normally be in C:\Users\your_name\Downloads, and then select "Extract without confirmation" or something else that extracts the zipped file into the current directory. Next, run the install.bat file.

  Next step: Try to release again and test on Windows. To install on Windows, use notepad to edit the install file, which opens a gui, then edit the destination directory to something that I don't need admin prvileges for. Mac stuff seems good now. Trying to get Windows good in all ways next, including with BioNetGen.

  For Windows, FROMDIR should be the current working directory. Also the DESTDIR thing doesn't work because it isn't being substituted correctly.
- Smoldyn distribution should be made much easier. For example, distribute pre-compiled Mac software, and add Smoldyn to MacPorts, Fink, etc. code databases. This has be done to some extent.

## Core Smoldyn

- Add a surface panel shape called a holey sphere. This would be a sphere but with as many holes as desired. It could serve as a neuron junction, the end of a dividing cell, etc. Another idea is to add a new surface feature for holes. These could be added to spheres, cylinders, triangles, etc. The simplest hole is a spherical hole; it says that the normal surface rules apply unless the molecule is within this sphere, in which case the surface effectively isn't there. This would be easy to define, easy to implement, and efficient, but would be a challenge for graphics.
- Movable internal surfaces. Lots of people have asked for this collection of new features. At a minimum, it should be possible to move surfaces using commands. Note that these moving commands should not use absolute coordinates but relative ones; this is so a repeated command would cause continuous motion. Eventually, surfaces should also move with diffusion coefficients, in response to specific molecules, etc. Also, deformable surfaces. A tremendous amount of work could be done here and would be very useful.

- Adaptive time steps.
- Check bimolecular reaction rates for surface-bound molecules.

## Graphics and I/O

- Off-screen rendering would be very helpful. Two options seem most promising. (1) Off-screen rendering with OpenGL. This requires a frame buffer object, which looks relatively straightforward but still has some challenges. (2) Dump the data to a text file that can be post-processed by VMD (visualization of molecular dynamics). For this, John Stone (VMD author) e-mailed me 3/31/11 to suggest that I use the XYZ file format for the molecules plus a VMD script for the surfaces.
- A minor but helpful feature would be the possibility of graphing molecule concentrations at the bottom of the graphics window.
- In `loadsim`, `output_files`, a file name that is identical to the config. file name should result in a warning and ask user if it's desired. (There is already a file overwrite warning, so this isn't really needed; it's also a bit harder to add than it seemed it might be.)
- Keypress 'i' should enter interactive mode in which the user can type commands into the standard input and they will be executed right away.
- There should be statements and Libsmoldyn functions for removing things from Smoldyn. Like removing surfaces, surface panels, reactions, molecules, etc. Species removal would be harder.

## Code improvements

- First order reactions would be more efficient with an event queue rather than a probabilistic likelihood at each time step.
- I think I see two ways to improve performance in `checksurfaces`: (1) instead of `for(bptr1=pos2box...)`, do `for(bptr1=mptr->box,...)`. (2) End this same box loop when `crossmin`$<2$.
- Speed for 3D systems can be improved by always checking for 3D first, rather than last.
- Various cleanups would be nice for reactions: (1) `doreact` could be streamlined slightly for order 2 reactions by precomputing $x$, (3) `rxnss->table` symmetry is performed by having separate identical sides, but one side could just as easily point to the same data as does the other side, (5) allostery may need improvement, (6) derive theory for non-one reaction probabilities, (7) implement unimolecular equilibrium constant reactions in which the user just enters the reactants, the equilibrium constant, and the time constant (which may be 0) and Smoldyn calculates transition probabilities.

## Major additions

- Fibers (such as DNA, actin, microtubules, MinD, FtsZ, etc.), fiber-bound molecules, etc. Also, membrane-bound polymers would be nice.
- Intrinsic molecular parameters. In this idea, the user enters fundamental intrinsic molecular parameters, and then Smoldyn calculates the model parameters from them. For example, the user enters molecular weight, and the occupied membrane area for surface-bound molecules, and Smoldyn calculates the diffusion coefficient from those things (see my diffusion coefficient rule-of-thumb in the MMB paper and the Saffman-Delbruck equation in PNAS 1975). Also, the user enters intrinsic reaction rates or activation energies, and ideal (short time step) binding radii, and Smoldyn calculates reaction rate constants from them. Ideally, the user could also specify viscosities for different surfaces and different compartments, and Smoldyn would adjust diffusion coefficients and binding radii automatically. Christine (Le Novere group) has already written some code that may be relevant to this idea. Having this feature would save the user from creating an explosion of molecular species for the various molecular environments. However, it might slow the program down substantially to check the environment for every potential dynamic aspect.

- Inclusion of continuous concentrations for chemical species that are abundant. Ideally, these concentrations should be updated with ODEs, PDEs, spatial- or non-spatial Langevin dynamics, or spatial- or non-spatial Gillespie algorithm, according to the user's choice.
- Derive theory for bimolecular reaction rates with probabilistic reactions.