Smoldyn Programmer's Manual
for Smoldyn version 2.29

Steve Andrews

©April, 2013

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. So far, nearly all of the development has been done by myself, Steve Andrews. However, this is starting to change. Nathan Addy is working on a rule-based modeling module, called Libmoleculizer, which is already distributed with the Smoldyn code and will hopefully be functional soon. Lorenzo Dematte has parallelized several of the core Smoldyn algorithms for graphical processing units (GPUs) and is integrating his parallel functions into the core Smoldyn code. And, Denis Gladkov is rewriting Smoldyn from scratch specifically for parallel GPU architectures. Yet others have expressed interest in adding other features.

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 → 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.

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
        pthreads
        Libmoleculizer
            Python
            libXML++
                libXML2
                    zlib
                    libiconv

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.

2.4 Building with CMake (versions 2.27 and higher)

Through version 2.26, Smoldyn built using the GNU Autoconf, Automake, and Libtool tools, using scripts originally written by Nathan Addy in 2009 and updated occasionally since then. Because these GNU tools are remarkably arcane (and because the Virtual Cell group uses CMake) I switched to CMake for versions 2.27 and afterwards. This transition will likely introduce building problems for a while, but I hope to get things working well as soon as possible.

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). For example, to compile with Libmoleculizer support, start CMake with "cmake .. -DOPTION_USE_LIBMZR=ON" (but, see below to actually make this work).

For compiling with Libmoleculizer, you first need to build Libmoleculizer. Change to the source/libmoleculizer-1.1.2/cmake directory, clean it out with "rm -r *", configure with "cmake .." and build with "make". If this doesn't work, then it's probably because you need to install libXML++. For that, see above. If you want, you can install Libmoleculizer using "sudo make install" but this is not necessary (and not advised). Once you have libmoleculizer built, return to the Smoldyn cmake directory and configure with "cmake .. -DOPTION_USE_LIBMZR=ON", followed by "make" and "sudo make install". This will re-make the libmoleculizer package, but it seems to work.

Following are some helpful build options:

option	effect
-DOPTION_USE_LIBMZR=ON	Build with Libmoleculizer module
-DOPTION_STATIC=ON	Build using static libraries
-DCMAKE_BUILD_TYPE=...	Choose CMake build type
options are: None, Debug, Release (default), RelWithDebInfo, and MinSizeRel	
-DOPTION_USE_OPENGL=OFF	Build without graphics support
-DOPTION_USE_LIBTIFF=OFF	Build without LibTiff support
-DOPTION_USE_ZLIB=OFF	Build without ZLib support
-OPTION_TARGET_SMOLDYN=OFF	Dont build stand-alone Smoldyn program
-DOPTION_TARGET_LIBSMOLDYN=ON	Build LibSmoldyn library

Versions 2.27 and 2.28 did not install Libmoleculizer correctly. This was fixed for version 2.29. However, at least so far, version 2.29 does not enable both Libmoleculizer and Libsmoldyn at once.

2.5   Building with GNU AutoTools (versions 2.26 and lower)

`./configure` When configuring, several options can be helpful. For all of them, enter them with the form: "`./configure --`*option1* `--`*option2*". There are a wide variety of standard options, which are easily found from the internet or by entering "`./configure --help`". For example, the "`--prefix` *directory*" option installs files to the specified file directory rather than default locations (see below). Other standard options enable setting of source directories, host operating systems, etc. In addition, Smoldyn offers the following custom options:

option	effect
--enable-libsmoldyn	compile and install Libsmoldyn
--enable-libmoleculizer	include Libmoleculizer in build

`--disable-opengl`	don't build for opengl
`--enable-threading`	build for POSIX threads (not recommended)

As an example, I compile the software in the `/Mac` directory using:

```
./configure --prefix=$HOME/SSA/Smoldyn/trunk/Mac --enable-libsmoldyn
```

You can also set environment variables before the configure statement, such as: "`CFLAGS="-O0 -g"` `./configure`". This is helpful for setting the compiler and compilation options. In particular, some helpful combinations include:

option	effect
`CFLAGS="-O0 -g"`	minimum optimizations; best for debugging
`CFLAGS="-O3 -g -pg"`	compile for profiling with gprof
`CFLAGS="-O3" CPPFLAGS="-O3"`	might improve running speed
`LIBS="-lglut -lGLU"`	might be needed to compile on Ubuntu systems

The configure.ac could be improved. However, I have not updated this file significantly because it seems to work and I don't want to break it. Also, it's hard to test because doing so requires a wide variety of computer systems with differing amounts of installed software. Nevertheless, this file is ready for an overhaul at some point.

make
   This does the actual code compiling of Smoldyn and other parts of the distribution (including Libmoleculizer if enabled, Libsmoldyn, SmolCrowd, and wrl2smol). This creates lots of object files in the source directory, several shared library files, and executable programs.

sudo make install
   This installs Smoldyn and other important files to their install locations. The default files and install locations (if not modified using the `--prefix` configure option) are:

path	file	file description
/usr/local/bin/	smoldyn	stand-alone Smoldyn program
/usr/local/bin/	SmolCrowd	utility to generate crowded spaces
/usr/local/bin/	wrl2smol	utility to convert VRML to Smoldyn
Libsmolydn files, if configured with `--enable-libsmoldyn`		
/usr/local/include/smoldyn	libsmoldyn.h	API for Libsmoldyn code library
/usr/local/include/smoldyn	smoldyn.h	Libsmoldyn structure declarations
/usr/local/include/smoldyn	smoldyn_config.h	Libsmoldyn configurations
/usr/local/lib/	libsmoldyn.la	Libsmoldyn for dynamic linking
/usr/local/lib/	libsmoldyn.0.dylib	Libsmoldyn for dynamic linking
/usr/local/lib/	libsmoldyn.dylib	alias for libsmoldyn.0.dylib
/usr/local/lib/	libsmoldyn.a	Libsmoldyn for static linking

Libmoleculizer, if it is enabled, is included in Smoldyn with the result that no Libmoleculizer files are installed to the system.

make clean
   This removes the build files, including object files and executables, from the source directory.

sudo make uninstall
   This removes all of the files that `make install` copied to their install locations.

make manual
   This is a non-standard make option. It simply copies the Smoldyn User Manual from the documentation

directory to the /trunk directory. This is used in the distribution process.

make dist

This makes tar and zip files for distribution. This makes smoldyn-2.25.tar.gz and smoldyn-2.25.zip (or whatever the current version number is). Note that the version number is defined at the top of the configure.ac file.

Cross-compiling for Windows

I compile Smoldyn for Windows by cross-compiling from Macintosh using the MinGW compiler. It took some effort to get things figured out here, but I think that it finally works. However, I do not include Libmoleculizer in Windows builds because it requires Python and other dependencies that I have not tried to address yet.

I downloaded and installed MinGW on a Mac Pro running OS 10.6 using MacPorts. It installed to `/opt/local/bin/` and works well. I use it in cross-compilation mode. AutoTools needs the host and build names. The host name is simply the MinGW name (i386-mingw32). To get the build name, I ran `./config.guess` and used that result, which was x86_64-apple-darwin10.8.0. CMake doesn't seem to need these names.

The MinGW download included an OpenGL library at `/opt/local/i386-mingw32/lib/libopengl32.a` and header files at `/opt/local/i386-mingw32/include/ GL/`. However, the download was missing both header and library files for the glut package. The OpenGL website offers glut version 3.7, which is the latest version, but dates back to 1998. Furthermore, that download offers glut.h, but not the library that I needed. Looking some more, I found that the website `http://www.-transmissionzero.-co.-uk/-computing-/ -using-glut-with-mingw/` says that the glut project has been replaced by the freeglut project and that this website offers the necessary freeglut downloads. Thus, I followed their advice. I downloaded freeglut and copied `glut.h, freeglut.h, freeglut_std.h, and freeglut_ext.h` into my MinGW GL include directory and copied `libfreeglut.a` into my MinGW lib directory.

Next, the MinGW download was also missing the LibTiff library. I looked for but couldn't find precompiled versions of LibTiff, so I instead got LibTiff 3.9.5 from `http://www.remotesensing.org/libtiff/`. I configured LibTiff with

```
CC="/opt/local/bin/i386-mingw32-gcc"./configure --prefix=$HOME/Desktop/tiffinstalltest/
--host=i386-mingw32 --build=x86_64-apple-darwin10.8.0
```

Then I entered `make` and `make install`. These yielded a lot of files in my tiffinstalltest directory. They included libtiff.a and various header files, plus a lot of stuff that I don't use, such as a lot of .exe files. I copied the header files to `/opt/local/i386-mingw32/include` and `libtiff.a` file to `/opt/local/i386-mingw32/lib`.

zlib is similar. I got zlib from `http://sourceforge.net/projects/mingw/files/MinGW/Extension/ zlib/zlib-1.2.5-4/`, but it was there as a .lzma file. To uncompress that, I had to get "The Unarchiver" from `http://code.google.com/p/theunarchiver/downloads/detail?name=TheUnarchiver2.7.1.zip\ &can=2\&q=`. Finally, I copied the zlib include and lib files to the MinGW include and lib directories.

Finally, I built Smoldyn for Windows. The following statements create a no-graphics version of just Smoldyn, and then a with-graphics version of both Smoldyn and Libsmoldyn. *Be sure to (1) enter* **make clean** *beforehand to remove prior object files, (2) watch out for LaTeX-specific backslashes, underscores, or other weirdness when copying and pasting the text below, (3) enter* **make install** *afterwards to copy the results to the* **windows** *directory, and (4) enter* **make clean** *afterwards to remove object files.*

```
CC="/opt/local/bin/i386-mingw32-gcc"./configure --prefix=$HOME/SSA/Smoldyn/trunk/windows
 --host=i386-mingw32 --build=x86_64-apple-darwin10.8.0 --disable-opengl
```

```
CC="/opt/local/bin/i386-mingw32-gcc"CFLAGS="-I/opt/local/i386-mingw32/include/GL -I/opt/
local/i386-mingw32/include"LDFLAGS="-L/opt/local/bin/i386-mingw32/lib -lopengl32 -lglu32
 -lfreeglut -ltiff"./configure --prefix=$HOME/SSA/Smoldyn/trunk/windows --host=i386-
mingw32 --build=x86_64-apple-darwin10.8.0 --enable-libsmoldyn
```

Cross-compiling for PowerPC Mac

I have also cross-compiled Smoldyn for a PowerPC Mac. The process worked smoothly, although I haven't heard yet whether the final result worked or not. I cross-compiled using:

```
./configure --prefix=$HOME/Desktop/ppc_smoldyn --build=x86_64-apple-darwin10.8.0 --host=
powerpc-*-darwin* --enable-libsmoldyn
```

2.6 Distribution files

This section applies to Smoldyn 2.26 and lower. Both this section and the distribution need to be updated to reflect the current CMake build system.

The Smoldyn distribution (for Mac or Linux) includes the following files and directories:

file	author/origin	description
acinclude.m4	Nathan, others	Autoconf macros, referenced from configure.ac
aclocal.m4	bootstrap.sh	Autoconf macros, referenced from configure.ac
compile	GNU Automake	wrapper for certain compilers
config.guess	bootstrap.sh	guesses the build system triplet
config.sub	bootstrap.sh	validates and canonicalizes build system triplet
configure	bootstrap.sh	shell script for determining compile settings
configure.ac	Nathan	Autoconf input file to make configure
COPYING GNU	GNU	general public license
depcomp	bootstrap.sh	some Automake thing
documentation	Steve	(directory) Smoldyn code documentation
examples	Steve	(directory) Smoldyn configuration files
INSTALL	bootstrap.sh	generic instructions for installing a GNU project
install-sh	bootstrap.sh	something to do with installing
ltmain.sh	bootstrap.sh	an Automake input file
Mac	Steve	Mac binaries and libraries
Makefile.am	Nathan	top level Automake makefile
Makefile.in	bootstrap.sh	a makefile
missing	bootstrap.sh	stub for missing GNU programs
README	Steve	extremely brief instructions
source	Steve, Nathan	(directory) all source code
windows	Steve	(directory) windows executables, dlls, libraries

Where the author is listed as bootstrap.sh, this means that the file was automatically generated through running the bootstrap.sh shell script, which actually generated the files with either Autoconf or Automake.

2.7 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

Libsmoldyn

3.1 About Libsmoldyn

Libsmoldyn is an interface to the Smoldyn simulator. At the moment, it can be accessed through C and C++, although I hope to have it accessible to other languages too, including Python, MatLab, and R. Libsmoldyn is complementary to the stand-alone Smoldyn program in that it is a little more difficult to use, but it provides much more flexibility. In addition, Libsmoldyn provides: (*i*) an application programming interface that will be relatively stable, even as Smoldyn is updated and improved, (*ii*) function names that are relatively sensible and that shouldn't collide with other function names in other software, and (*iii*) reasonably thorough error checking in every function which helps ensure that the user is using the function in a sensible way and in a way that won't crash Smoldyn. Libsmoldyn, as described here, is a strictly C++ interface. Using Swig, I'm creating Libsmoldyn bindings to other languages.

Libsmoldyn only barely supports graphics at present due to constraints imposed by the glut code library. This will be improved in a future version (by changing to the freeglut library, which doesn't insist on controlling the main event loop, as the glut library does).

3.2 Compiling Libsmoldyn

I changed build systems from GNU AutoTools to CMake in version 2.27. However, I did not update the Libsmoldyn part of the build system until version 2.29.

Using the GNU AutoTools build system (now obsolete for Smoldyn), Libsmoldyn compiled with the same build system as the stand-alone Smoldyn program. Prior advice follows. Simply enable Libsmoldyn by starting the build process with `./configure --enable-libsmoldyn`. Then, enter `make` and `sudo make install` as usual. This builds both static and dynamic Libsmoldyn libraries. Their default install locations are `/usr/local/lib` for the libraries and `/usr/local/include/smoldyn` for the header files.

3.3 Linking to Libsmoldyn

To enable a C program to use Libsmoldyn, add the line `#include <smoldyn/libsmoldyn.h>` to the top of your source code. After this, you should be able to compile your code. For example, in the `examples/S97_libsmoldyn/testcode/` directory, you'll find the testcode.c program. To compile this source code to object code, enter:

```
gcc -Wall -O0 -g -c testcode.c
```

The compile flags `-O0 -g` aren't necessary but can be useful for debugging purposes. If compiling doesn't work at this stage, it's probably because you're missing the header files. Make sure that you have libsmoldyn.h, smoldyn.h, and smoldyn_config.h in the `/usr/local/include` directory.

Next, this object code needs to be linked to the Libsmoldyn library. It can be linked statically, meaning that the Libsmoldyn code will be copied into the final result, or it can be linked dynamically, so that

the final result will simply reference the Libsmoldyn code that is stored separately. Either way, what you enter for the linking depends on whether you compiled Libsmoldyn to use OpenGL (configure options were simply `--enable libsmoldyn`) or not to use OpenGL (configure options were `--enable-libsmoldyn --disable-opengl`). The linking steps for the testcode program are shown below for these four options. The options that use OpenGL assume that you're on a Macintosh; for other systems, the result is a little simpler, but I'm not sure what it is.

Static link, no OpenGL:

```
gcc testcode.o /usr/local/lib/libsmoldyn.a -o testcode
```

Static link, with OpenGL:

```
gcc testcode.o /usr/local/lib/libsmoldyn.a -I/System/Library/Frameworks/OpenGL.framework
/Headers -I/System/Library/Frameworks/GLUT.framework/Headers -framework GLUT -framework
OpenGL -framework Cocoa -L/System/Library/Frameworks/OpenGL.framework/Libraries -o
testcode -ltiff
```

Dynamic link, no OpenGL:

```
gcc testcode.o -o testcode -lsmoldyn
```

Dynamic link, with OpenGL:

```
gcc testcode.o -I/System/Library/Frameworks/OpenGL.framework/Headers -I/System/Library/
Frameworks/GLUT.framework/Headers -framework GLUT -framework OpenGL -framework Cocoa -L/
System/Library/Frameworks/OpenGL.framework/Libraries -o testcode -lsmoldyn -ltiff
```

3.4   Libsmoldyn error trapping

Every function in Libsmoldyn checks that its input values are acceptable and also that no errors arise in the function execution. These errors are returned to the host library in a number of ways. Most Libsmoldyn functions (e.g. `smolRunSim`) return any error codes directly, which makes it easy to see if an error arose. However, a few functions (e.g. `smolNewSim`) return other types of values and so return some other indication of success or failure (e.g. `NULL`). In addition, some functions can raise warnings, which indicate that behavior is unusual but not incorrect.

For all of these errors and warnings, get the details of the problem using the function `smolGetError`, which will return the error code, the name of the function where the error arose, and a descriptive error string. This will also clear the error, if desired. If errors are not cleared, they are left until they are overwritten by subsequent errors. Warnings are also left until they are cleared or overwritten.

When writing code, it can be helpful to put Libsmoldyn into its debugging mode using the `smolSetDebugMode` function. Doing this causes any errors that arise to be displayed to stderr.

The possible error codes are declared in libsmoldyn.h with:

```
enum ErrorCode {enum ErrorCode {ECok=0,ECnotify=-1,ECwarning=-2,ECnonexist=-3,
  ECall=-4,ECmissing=-5,ECbounds=-6,ECsyntax=-7,ECerror=-8,ECmemory=-9,ECbug=-10,
  ECsame=-11};
```

Their interpretations are:

value	code	interpretation
0	ECok	no error
-1	ECnotify	message about correct behavior
-2	ECwarning	unusual but not incorrect behavior
-3	ECnonexist	a function input specifies an item that doesn't exist
-4	ECsame	error code should be unchanged from a prior code
-5	ECall	an argument of "all" was found and may not be permitted
-6	ECmissing	a necessary function input parameter is missing
-7	ECbounds	a function input parameter is out of bounds

-8	`ECsyntax`	function inputs don't make syntactical sense
-9	`ECerror`	unspecified error condition
-10	`ECmemory`	Smoldyn was unable to allocate the necessary memory
-11	`ECbug`	error arose which should not have been possible

Error checking system internal to libsmoldyn.c

This subsection describes how to write Libsmoldyn functions using error checking. While it is an essential part of all Libsmoldyn functions, these details are not important for most Libsmoldyn users.

1. The first line of every Libsmoldyn function should be `const char *funcname="`*function_name*`";`. This name will be returned with any error message to tell the user where the error arose.

2. Within the function, check for warnings or errors with the `LCHECK` macro. The macro format is `LCHECK(`*condition*`,funcname,`*error_code*`,"`*message*`");`. This checks that the test *condition* is true, and issues a notification, warning, or error when this is not the case. The *message* should be a descriptive message that is under 256 characters in length.

3. Most functions return an "`enum ErrorCode`". If this is the case for your function, and your function might return a notification and/or a warning, then end the main body of the function with `return libwarncode;`. If it cannot return a notification or a warning, then end it with `return ECok;`. Finally, if it does not return an "`enum ErrorCode`", then it needs to return some other error condition that will tell the user to check for errors using `smolGetError`.

4. After the main body of the function, add a goto target called `failure:`.

5. Assuming the function returns an "`enum ErrorCode`", end the function with `return liberrorcode;`.

The `smolSetTimeStep` function provides an excellent and simple example of how Libsmoldyn functions typically address errors. It is:

```
enum ErrorCode smolSetTimeStep ( simptr sim , double timestep ) {
  const char *funcname ="smolSetTimeStep";

  LCHECK ( sim , funcname , ECmissing ,"missing␣sim");
  LCHECK ( timestep >0 , funcname , ECbounds ,"timestep␣is␣not␣>␣0");
  simsettime ( sim , timestep ,3);
  return ECok;
 failure :
  return liberrorcode; }
```

The `smolGet...Index` functions are worth a comment. Each of these functions returns the index of an item, such as a species or a surface, based on the name of the item. If the name is not found or other errors arise, then these functions return the error code, cast as an integer. Also, if the name is "all", then these functions return the error code `ECall` and set the error string "species cannot be 'all'", or equivalent. A typical use of these functions is seen in `smolSetSpeciesMobility`, which includes the following code:

```
i = smolGetSpeciesIndex ( sim , species );
if ( i ==( int ) ECall ) smolClearError ();
else LCHECK ( i >0 , funcname , ECsame , NULL );
```

In this particular case, this function permits an input of "all", so it clears errors that arise from this return value, and leaves `i` as a negative value for later use.

3.5 Libsmoldyn quick function guide

The Libsmoldyn functions correspond relatively closely to the Smoldyn language statements, although not perfectly. However, all functionality should be available using either method. The following table lists the correspondences. Statements preceded by asterisks need to be either entered in statement blocks or preceded by the statement's context (e.g. with `surface` *name*). Where correspondence does not apply, the table lists "N/A".

Statement	Libsmoldyn function
About the input	
#	N/A
/* ... */	N/A
read_file	`smolLoadSimFromFile`, `smolReadConfigString`
end_file	N/A
define	N/A
define_global	N/A
undefine	N/A
ifdefine	N/A
ifundefine	N/A
else	N/A
endif	N/A
display_define	N/A
N/A	`smolSetError`
N/A	`smolGetError`
N/A	`smolClearError`
N/A	`smolSetDebugMode`
N/A	`smolErrorCodeToString`
Space and time	
dim	`smolNewSim`
boundaries	`smolNewSim`, `smolSetBoundaryType`
low_wall	`smolNewSim`, `smolSetBoundaryType`
high_wall	`smolNewSim`, `smolSetBoundaryType`
time_start	`smolSetSimTimes`, `smolSetTimeStart`
time_stop	`smolSetSimTimes`, `smolSetTimeStop`
time_step	`smolSetSimTimes`, `smolSetTimeStep`
time_now	`smolSetTimeNow`
Molecules	
species	`smolAddSpecies`
N/A	`smolGetSpeciesIndex`
N/A	`smolGetSpeciesName`
difc	`smolSetSpeciesMobility`
difm	`smolSetSpeciesMobility`
drift	`smolSetSpeciesMobility`
mol	`smolAddSolutionMolecules`
surface_mol	`smolAddSurfaceMolecules`
compartment_mol	`smolAddCompartmentMolecules`
molecule_lists	`smolAddMolList`
mol_list	`smolAddSpecies`, `smolSetMolList`
N/A	`smolGetMolListIndex`
N/A	`smolGetMolListName`
max_mol	`smolSetMaxMolecules`
N/A	`smolGetMoleculeCount`
Graphics	
graphics	`smolSetGraphicsParams`

graphic_iter	`smolSetGraphicsParams`
graphic_delay	`smolSetGraphicsParams`
frame_thickness	`smolSetFrameStyle`
frame_color	`smolSetFrameStyle`
grid_thickness	`smolSetGridStyle`
grid_color	`smolSetGridStyle`
background_color	`smolSetBackgroundStyle`
display_size	`smolSetMoleculeStyle`
color	`smolSetMoleculeStyle`
tiff_iter	`smolSetTiffParams`
tiff_name	`smolSetTiffParams`
tiff_min	`smolSetTiffParams`
tiff_max	`smolSetTiffParams`
light	`smolSetLightParams`
text_color	`smolSetTextStyle`
text_display	`smolAddTextDisplay`

Run-time commands	
output_root	`smolSetOutputPath`
output_files	`smolAddOutputFile`
append_files	`smolAddOutputFile`
output_file_number	`smolAddOutputFile`
cmd	`smolAddCommand, smolAddCommandFromString`

Surfaces	
start_surface	`smolAddSurface`
new_surface	`smolAddSurface`
* name	`smolAddSurface`
N/A	`smolGetSurfaceIndex`
N/A	`smolGetSurfaceName`
* action	`smolSetSurfaceAction`
* rate	`smolSetSurfaceRate`
* rate_internal	`smolSetSurfaceRate`
* color	`smolSetSurfaceFaceStyle, smolSetSurfaceEdgeStyle`
* thickness	`smolSetSurfaceEdgeStyle`
* stipple	`smolSetSurfaceEdgeStyle`
* polygon	`smolSetSurfaceFaceStyle`
* shininess	`smolSetSurfaceFaceStyle`
* panel	`smolAddPanel`
N/A	`smolGetPanelIndex`
N/A	`smolGetPanelName`
* jump	`smolSetPanelJump`
* neighbors	`smolAddPanelNeighbor`
* unbounded_emitter	`smolAddSurfaceUnboundedEmitter`
* end_surface	N/A
epsilon	`smolSetSurfaceSimParams`
margin	`smolSetSurfaceSimParams`
neighbor_dist	`smolSetSurfaceSimParams`

Compartments	
start_compartment	`smolAddCompartment`
new_compartment	`smolAddCompartment`
* name	`smolAddCompartment`
N/A	`smolGetCompartmentIndex`
N/A	`smolGetCompartmentName`
* surface	`smolAddCompartmentSurface`
* point	`smolAddCompartmentPoint`

* compartment	smolAddCompartmentLogic
* end_compartment	N/A
Reactions	
reaction	smolAddReaction
N/A	smolGetReactionIndex
N/A	smolGetReactionName
reaction_cmpt	smolSetReactionRegion
reaction_surface	smolSetReactionRegion
reaction_rate	smolAddReaction, smolSetReactionRate
confspread_radius	smolSetReactionRate
binding_radius	smolSetReactionRate
reaction_probability	smolSetReactionRate
reaction_production	smolSetReactionRate
reaction_permit	not supported
reaction_forbid	not supported
product_placement	smolSetReactionProducts
Ports	
start_port	smolAddPort
new_port	smolAddPort
* name	smolAddPort
N/A	smolGetPortIndex
N/A	smolGetPortName
* surface	smolAddPort
* face	smolAddPort
* end_port	N/A
N/A	smolAddPortMolecules
N/A	smolGetPortMolecules
Simulation settings	
rand_seed	smolSetRandomSeed
accuracy	not supported
molperbox	smolSetPartitions
boxsize	smolSetPartitions
gauss_table_size	not supported
epsilon	smolSetSurfaceSimParams
margin	smolSetSurfaceSimParams
neighbor_dist	smolSetSurfaceSimParams
pthreads	not supported
Libsmoldyn actions	
N/A	smolUpdateSim
N/A	smolRunTimeStep
N/A	smolRunSim
N/A	smolRunSimUntil
N/A	smolFreeSim
N/A	smolDisplaySim
N/A	smolPrepareSimFromFile

3.6   Libsmoldyn header file

Following is the entire Libsmoldyn header file, libsmoldyn.h. This lists all of the function declarations. If there is a discrepancy between declarations listed here and those listed in following sections, the ones shown here are almost certainly the correct ones. This file references smoldyn.h, which lists all of the data structure declarations and enumerated type definitions.

   If you compiled and installed Smoldyn using the default configuration, both files should be in your

/usr/local/include/smoldyn directory. Also in this directory is the smoldyn_config.h file. This file was used for compiling Smoldyn and Libsmoldyn but is not needed afterwards. Nevertheless, it's copied to the /usr/local/include/smoldyn directory so that programs that call Libsmoldyn can know what options Libsmoldyn was built with.

```
/* Steven Andrews, started 10/22/2001.
 This is an application programming interface for the Smoldyn program.  See
 documentation called Smoldyn_doc1.pdf and Smoldyn_doc2.pdf.
 Copyright 2003-2011 by Steven Andrews.  This work is distributed under the terms
 of the Gnu General Public License (GPL). */

#ifndef __libsmoldyn_h__
#define __libsmoldyn_h__

/* The following Swig directives are only read by the swig program */
#ifdef SWIG
%module libsmoldyn
%{
#define SWIG_FILE_WITH_INIT
#include "libsmoldyn.h"
%}
#endif


#include "smoldyn.h"

enum ErrorCode {ECok=0,ECnotify=-1,ECwarning=-2,ECnonexist=-3,ECall=-4,ECmissing
    =-5,ECbounds=-6,ECsyntax=-7,ECerror=-8,ECmemory=-9,ECbug=-10,ECsame=-11,
    ECwildcard=-12};

#ifdef __cplusplus
extern "C" {
#endif

/***************************** Miscellaneous ********************************/

double        smolGetVersion(void);

/******************************** Errors ************************************/

void          smolSetLogging(FILE *logfile,void (*logFunction)(simptr,int,const
    char*,...));
void          smolSetThrowing(int corethreshold,int libthreshold);
void          smolSetError(const char *errorfunction,enum ErrorCode errorcode,
    const char *errorstring);
void          smolSetErrorNT(const char *errorfunction,enum ErrorCode errorcode,
    const char *errorstring);
enum ErrorCode smolGetError(char *errorfunction,char *errorstring,int clearerror);
void          smolClearError(void);
void          smolSetDebugMode(int debugmode);
char*         smolErrorCodeToString(enum ErrorCode erc,char *string);

/***************************** Sim structure ********************************/

simptr        smolNewSim(int dim,double *lowbounds,double *highbounds);
enum ErrorCode smolUpdateSim(simptr sim);
enum ErrorCode smolRunTimeStep(simptr sim);
enum ErrorCode smolRunSim(simptr sim);
enum ErrorCode smolRunSimUntil(simptr sim,double breaktime);
```

```
enum ErrorCode smolFreeSim(simptr sim);
enum ErrorCode smolDisplaySim(simptr sim);


/************************* Read configuration file **************************/


simptr         smolPrepareSimFromFile(const char *filepath,const char *filename,
    const char *flags);
enum ErrorCode smolLoadSimFromFile(const char *filepath,const char *filename,
    simptr *simpointer,const char *flags);
enum ErrorCode smolReadConfigString(simptr sim,const char *statement,char *
    parameters);


/**************************** Simulation settings ***************************/


enum ErrorCode smolSetSimTimes(simptr sim,double timestart,double timestop,double
    timestep);
enum ErrorCode smolSetTimeStart(simptr sim,double timestart);
enum ErrorCode smolSetTimeStop(simptr sim,double timestop);
enum ErrorCode smolSetTimeNow(simptr sim,double timenow);
enum ErrorCode smolSetTimeStep(simptr sim,double timestep);
enum ErrorCode smolSetRandomSeed(simptr sim,long int seed);
enum ErrorCode smolSetPartitions(simptr sim,const char *method,double value);


/******************************* Graphics ***********************************/


enum ErrorCode smolSetGraphicsParams(simptr sim,char *method,int timesteps,double
    delay);
enum ErrorCode smolSetTiffParams(simptr sim,int timesteps,const char *tiffname,int
     lowcount,int highcount);
enum ErrorCode smolSetLightParams(simptr sim,int lightindex,double *ambient,double
     *diffuse,double *specular,double *position);
enum ErrorCode smolSetBackgroundStyle(simptr sim,double *color);
enum ErrorCode smolSetFrameStyle(simptr sim,double thickness,double *color);
enum ErrorCode smolSetGridStyle(simptr sim,double thickness,double *color);
enum ErrorCode smolSetTextStyle(simptr sim,double *color);
enum ErrorCode smolAddTextDisplay(simptr sim,char *item);


/**************************** Runtime commands ******************************/


enum ErrorCode smolSetOutputPath(simptr sim,const char *path);
enum ErrorCode smolAddOutputFile(simptr sim,char *filename,int suffix,int append);
//?? needs function for setting output precision
enum ErrorCode smolAddCommand(simptr sim,char type,double on,double off,double
    step,double multiplier,const char *commandstring);
enum ErrorCode smolAddCommandFromString(simptr sim,char *string);


/****************************** Molecules **********************************/


enum ErrorCode smolAddSpecies(simptr sim,const char *species,const char *mollist);
int            smolGetSpeciesIndex(simptr sim,const char *species);
int            smolGetSpeciesIndexNT(simptr sim,const char *species);
char*          smolGetSpeciesName(simptr sim,int speciesindex,char *species);
enum ErrorCode smolSetSpeciesMobility(simptr sim,const char *species,enum
    MolecState state,double difc,double *drift,double *difmatrix);
//?? needs function smolSetSpeciesSurfaceDrift
enum ErrorCode smolAddMolList(simptr sim,const char *mollist);
int            smolGetMolListIndex(simptr sim,const char *mollist);
int            smolGetMolListIndexNT(simptr sim,const char *mollist);
char*          smolGetMolListName(simptr sim,int mollistindex,char *mollist);
```

```
enum ErrorCode smolSetMolList(simptr sim,const char *species,enum MolecState state
    ,const char *mollist);
enum ErrorCode smolSetMaxMolecules(simptr sim,int maxmolecules);
enum ErrorCode smolAddSolutionMolecules(simptr sim,const char *species,int number,
    double *lowposition,double *highposition);
enum ErrorCode smolAddCompartmentMolecules(simptr sim,const char *species,int
    number,const char *compartment);
enum ErrorCode smolAddSurfaceMolecules(simptr sim,const char *species,enum
    MolecState state,int number,const char *surface,enum PanelShape panelshape,
    const char *panel,double *position);
int            smolGetMoleculeCount(simptr sim,const char *species,enum MolecState
     state);
enum ErrorCode smolSetMoleculeStyle(simptr sim,const char *species,enum MolecState
     state,double size,double *color);

/******************************** Surfaces ********************************/

enum ErrorCode smolSetBoundaryType(simptr sim,int dimension,int highside,char type
    );
enum ErrorCode smolAddSurface(simptr sim,const char *surface);
int            smolGetSurfaceIndex(simptr sim,const char *surface);
int            smolGetSurfaceIndexNT(simptr sim,const char *surface);
char*          smolGetSurfaceName(simptr sim,int surfaceindex,char *surface);
enum ErrorCode smolSetSurfaceAction(simptr sim,const char *surface,enum PanelFace
    face,const char *species,enum MolecState state,enum SrfAction action);
enum ErrorCode smolSetSurfaceRate(simptr sim,const char *surface,const char *
    species,enum MolecState state,enum MolecState state1,enum MolecState state2,
    double rate,const char *newspecies,int isinternal);
enum ErrorCode smolAddPanel(simptr sim,const char *surface,enum PanelShape
    panelshape,const char *panel,const char *axisstring,double *params);
int            smolGetPanelIndex(simptr sim,const char *surface,enum PanelShape *
    panelshapeptr,const char *panel);
int            smolGetPanelIndexNT(simptr sim,const char *surface,enum PanelShape
    *panelshapeptr,const char *panel);
char*          smolGetPanelName(simptr sim,const char *surface,enum PanelShape
    panelshape,int panelindex,char *panel);
enum ErrorCode smolSetPanelJump(simptr sim,const char *surface,const char *panel1,
    enum PanelFace face1,const char *panel2,enum PanelFace face2,int
    isbidirectional);
enum ErrorCode smolAddSurfaceUnboundedEmitter(simptr sim,const char *surface,enum
    PanelFace face,const char *species,double emitamount,double *emitposition);
enum ErrorCode smolSetSurfaceSimParams(simptr sim,const char *parameter,double
    value);
enum ErrorCode smolAddPanelNeighbor(simptr sim,const char *surface1,const char *
    panel1,const char *surface2,const char *panel2,int reciprocal);
enum ErrorCode smolSetSurfaceStyle(simptr sim,const char *surface,enum PanelFace
    face,enum DrawMode mode,double thickness,double *color,int stipplefactor,int
    stipplepattern,double shininess);

/****************************** Compartments ******************************/

enum ErrorCode smolAddCompartment(simptr sim,const char *compartment);
int            smolGetCompartmentIndex(simptr sim,const char *compartment);
int            smolGetCompartmentIndexNT(simptr sim,const char *compartment);
char*          smolGetCompartmentName(simptr sim,int compartmentindex,char *
    compartment);
enum ErrorCode smolAddCompartmentSurface(simptr sim,const char *compartment,const
    char *surface);
enum ErrorCode smolAddCompartmentPoint(simptr sim,const char *compartment,double *
```

```
   point);
enum ErrorCode smolAddCompartmentLogic(simptr sim,const char *compartment,enum
    CmptLogic logic,const char *compartment2);


/******************************* Reactions ********************************/


enum ErrorCode smolAddReaction(simptr sim,const char *reaction,const char *
    reactant1,enum MolecState rstate1,const char *reactant2,enum MolecState rstate2
    ,int nproduct,const char **productspecies,enum MolecState *productstates,double
     rate);
int          smolGetReactionIndex(simptr sim,int *orderptr,const char *reaction)
    ;
int          smolGetReactionIndexNT(simptr sim,int *orderptr,const char *
    reaction);
char*        smolGetReactionName(simptr sim,int order,int reactionindex,char *
    reaction);
enum ErrorCode smolSetReactionRate(simptr sim,const char *reaction,double rate,int
     type);
enum ErrorCode smolSetReactionRegion(simptr sim,const char *reaction,const char *
    compartment,const char *surface);
enum ErrorCode smolSetReactionProducts(simptr sim,const char *reaction,enum
    RevParam method,double parameter,const char *product,double *position);


/********************************* Ports **********************************/


enum ErrorCode smolAddPort(simptr sim,const char *port,const char *surface,enum
    PanelFace face);
int          smolGetPortIndex(simptr sim,const char *port);
int          smolGetPortIndexNT(simptr sim,const char *port);
char*        smolGetPortName(simptr sim,int portindex,char *port);
enum ErrorCode smolAddPortMolecules(simptr sim,const char *port,int nmolec,const
    char *species,double **positions);
int          smolGetPortMolecules(simptr sim,const char *port,const char *
    species,enum MolecState state,int remove);

#ifdef __cplusplus
}
#endif


#endif
```

3.7   Libsmoldyn functions

General comments

None of the functions allocate memory, except within the simulation data structure. This means, for example, that all functions that return strings do not allocate these strings themselves, but instead write the string text to memory that the library user allocated and gave to the function. All strings are fixed at STRCHAR characters, where this constant is defined in string2.h to 256 characters.

Miscellaneous

```
double smolGetVersion(void);
```
   Returns the Smoldyn version number.

Errors

`void smolSetError(const char *errorfunction,enum ErrorCode errorcode,const char *errorstring);`
> This function is probably not useful for most users. Sets the Libsmoldyn error code to `errorcode`, error function to `errorfunction`, and error string to `errorstring`. The sole exception is if `errorcode` is `ECsame` then this does nothing and simply returns. Back to it's normal operation, this also either sets or clears the Libsmoldyn warning code, as appropriate. If `errorstring` is entered as `NULL`, this clears the current error string, and similarly for `errorfunction`.

`enum ErrorCode smolGetError(char *errorfunction,char *errorstring,int clearerror);`
> Returns the current LibSmoldyn error code directly, returns the function where the error occurred in `errorfunction` if it is not `NULL`, and returns the error string in `errorstring` if it is not `NULL`. Set `clearerror` to 1 to clear the error and 0 to leave any error condition unchanged.

`void smolClearError(void);`
> Clears any error condition.

`void smolSetDebugMode(int debugmode);`
> Enter `debugmode` as 1 to enable debugging and 0 to disable debugging. When debug mode is turned on, all errors are displayed to stderr, as are all cleared errors. By turning on debug mode, you can often avoid checking for errors with additional code and you also typically don't need to call `smolGetError`.

`char* smolErrorCodeTostring(enum ErrorCode erc,char *string);`
> Returns a string both directly and in `string` that corresponds to the error code in `erc`. For example, if `erc` is `ECmemory`, this returns the string "memory".

Sim structure

`simptr smolNewSim(int dim,double *lowbounds,double *highbounds);`
> Creates and returns a new sim structure. The structure is initialized for a `dim` dimensional system that has boundaries defined by the points `lowbounds` and `highbounds`. Boundaries are transmitting (modify them with `smolSetBoundaryType`). Returns `NULL` upon failure.

`enum ErrorCode smolUpdateSim(simptr sim);`
> Updates the simulation structure. This calculates all simulation parameters from physical parameters, sorts lists, and generally does everything required to make a simulation ready to run. It may be called multiple times.

`enum ErrorCode smolRunTimeStep(simptr sim);`
> Runs one time step of the simulation. Returns an error if the simulation terminates unexpectedly during this time step or a warning if it terminates normally.

`enum ErrorCode smolRunSim(simptr sim);`
> Runs the simulation until it terminates. Returns an error if the simulation terminates unexpectedly during this time step or a warning if it terminates normally.

`enum ErrorCode smolRunSimUntil(simptr sim,double breaktime);`
> Runs the simulation either until it terminates or until the simulation time equals or exceeds `breaktime`.

`enum ErrorCode smolFreeSim(simptr sim);`
> Frees the simulation data structure.

`enum ErrorCode smolDisplaySim(simptr sim);`
> Displays all relevant information about the simulation system to stdout.

Read configuration file

`simptr smolPrepareSimFromFile(char *filepath,char *filename,char *flags);`

Reads the Smoldyn configuration file that is at `filepath` and has file name `filename`, sets it up, and outputs simulation diagnostics to stdout. Returns the sim structure, or `NULL` if an error occurred. `flags` are the command line flags that are entered for normal Smoldyn use. Either or both of `filepath` and `flags` can be sent in as `NULL` if there is nothing to report. After this function runs successfully, it should be possible to call `smolRunSim` or `smolRunTimeStep`.

`enum ErrorCode smolLoadSimFromFile(char *filepath,char *filename,simptr *simpointer,char *flags);`

Loads part or all of a sim structure from the file that is at `filepath` and has file name `filename`. Send in `simpointer` as a pointer to sim, where sim may be an existing simulation structure that this function will append or `NULL` if it is to be created by this function. `flags` are the command line flags that are entered for normal Smoldyn use. Either or both of `filepath` and `flags` can be sent in as `NULL` if there is nothing to report. After this function runs successfully, call `smolUpdateSim` to calculate simulation parameters.

`enum ErrorCode smolReadConfigString(simptr sim,char *statement,char *parameters);`

Reads and processes what would normally be a single line of a configuration file. The first word of the line is the statement name, entered here as `statement`, while the rest of the line is entered as `parameters`. Separate different parameters with spaces. The same parser is used as for normal Smoldyn configuration files. This function does not make use of block style input formatting, such as for surface definitions. This means that a new surface needs to declared with "`new_surface` *name*" and all subsequent surface definitions need to start with "`surface` *name*". Analogous rules apply to compartments and port.

Simulation settings

`enum ErrorCode smolSetSimTimes(simptr sim,double timestart,double timestop,double timestep);`

Sets all of the simulation time parameters to the values entered here. In addition the simulation "time now" is set to `timestart`.

`enum ErrorCode smolSetTimeStart(simptr sim,double timestart);`

Sets the simulation starting time.

`enum ErrorCode smolSetTimeStop(simptr sim,double timestop);`

Sets the simulation stopping time.

`enum ErrorCode smolSetTimeNow(simptr sim,double timenow);`

Sets the simulation current time.

`enum ErrorCode smolSetTimeStep(simptr sim,double timestep);`

Sets the simulation time step, which must be greater than 0.

`enum ErrorCode smolSetRandomSeed(simptr sim,double seed);`

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.

`enum ErrorCode smolSetPartitions(simptr sim,char *method,double value);`

Sets the virtual partitions in the simulation volume. Enter `method` as "molperbox" and then enter `value` with the requested number of molecules per partition volume; the default, which is used if this function is not called at all, is a target of 4 molecules per box. Or, enter `method` as "boxsize" and enter `value` with the requested partition spacing. In this latter case, the actual partition spacing may be larger or smaller than the requested value in order to fit an integer number of partitions into each coordinate of the simulation volume.

Graphics

`enum ErrorCode smolSetGraphicsParams(simptr sim,char *method,int timesteps,double delay);`
> Sets basic simulation graphics parameters. Enter `method` as "none" for no graphics (the default), "opengl" for fast but minimal OpenGL graphics, "opengl_good" for improved OpenGL graphics, "opengl_better" for fairly good OpenGL graphics, or as `NULL` to not set this parameter currently. Enter `timesteps` with a positive integer to set the number of simulation time steps between graphics renderings (1 is the default) or with a negative number to not set this parameter currently. Enter `delay` as a non-negative number to set the minimum number of milliseconds that must elapse between subsequent graphics renderings in order to improve visualization (0 is the default) or as a negative number to not set this parameter currently.

`enum ErrorCode smolSetTiffParams(simptr sim,int timesteps,char *tiffname,int lowcount,int highcount);`
> Sets parameters for the automatic collection of TIFF format snapshots of the graphics window. `timesteps` is the number of simulation timesteps that should elapse between subsequent snapshots, `tiffname` is the root filename of the output TIFF files, `lowcount` is a number that is appended to the filename of the first snapshot and which is then incremented for subsequent snapshots, and `highcount` is the last numbered file that will be collected. Enter negative numbers for `timesteps`, `lowcount`, and/or `highcount` to not set these parameters, and enter `NULL` for `tiffname` to not set the file name.

`enum ErrorCode smolSetLightParams(simptr sim,int lightindex,double *ambient,double *diffuse,double *specular,double *position);`
> Sets the lighting parameters that are used for the rendering method "opengl_better". Enter `lightindex` as -1 for the global ambient light (in which case `diffuse`, `specular`, and `position` should all be NULL) or as 0 to 8 for one of the 8 light sources. For each light source, you can specify the 4-value color vector for the light's ambient, diffuse, and specular properties (all values should be between 0 and 1). You can also specify the 3-dimensional position for the light. To not set a property, just enter the respective vector as `NULL`.

`enum ErrorCode smolSetBackgroundStyle(simptr sim,double *color);`
> Sets the color of the graphics display background. `color` is a 4-value vector with red, green, blue, and alpha values.

`enum ErrorCode smolSetFrameStyle(simptr sim,double thickness,double *color);`
> Sets the thickness and the color of the wire frame that outlines the simulation system in the graphics window. Enter `thickness` as 0 for no frame, as a positive number for the number of points in thickness, or as a negative number to not set this parameter. Enter `color` as a 4-value vector with the frame color, or as `NULL` to not set it.

`enum ErrorCode smolSetGridStyle(simptr sim,double thickness,double *color);`
> Sets the thickness and the color of a grid that shows where the partitions are that separate Smoldyn's virtual boxes. Enter `thickness` as 0 for no grid, as a positive number for the number of points in thickness, or as a negative number to not set this parameter. Enter `color` as a 4-value vector with the grid color, or as `NULL` to not set it.

`enum ErrorCode smolSetTextStyle(simptr sim,double *color);`
> Sets the color of any text that is displayed to the graphics window. `color` is a 4-value vector with red, green, blue, and alpha values.

`enum ErrorCode smolAddTextDisplay(simptr sim,char *item);`
> Adds `item` to the list of things that Smoldyn should display as text to the graphics window. Currently supported options are "time" and the names of species and, optionally, their states. For species and states, the graphics window shows the number of molecules.

Runtime commands

`enum ErrorCode smolSetOutputPath(simptr sim,char *path);`
> Sets the file path for text output files to `path`.

`enum ErrorCode smolAddOutputFile(simptr sim,char *filename,int suffix,int append);`
> Declares the file called `filename` as a file for output by one or more runtime commands. Note that spaces are not permitted in the file name. If `suffix` is non-negative, then the file name is suffixed by this integer, which can be helpful for creating output file stacks. Enter `append` as 1 if any current file should simply be appended, or to 0 if any current file should be overwritten.

`enum ErrorCode smolAddCommand(simptr sim,char type,double on,double off,double`
`step,double multiplier,char *commandstring);`
> Adds a run-time command to the simulation, including its timing instructions. This function should generally be called after `smolSetSimTimes` to make sure that command times get set correctly. The following table lists the command type options along with the other parameters that are used for each type. Parameters that are not required are simply ignored. The `commandstring` is the command name followed by any command parameters.

`type`	meaning	`on`	`off`	`step`	`multiplier`
Continuous time queue					
b	before simulation	-	-	-	-
a	after simulation	-	-	-	-
@	at fixed time	time	-	-	-
i	fixed intervals	time on	time off	time step	-
x	exponential intervals	time on	time off	min. time step	multiplier
Integer time queue					
B	before simulation	-	-	-	-
A	after simulation	-	-	-	-
&	at fixed iteration	iteration	-	-	-
I	fixed iteration intervals	iter. on	iter. off	iter. step	-
E	every time step	-	-	-	-
N	every n'th time step	-	-	iter. step	-

`enum ErrorCode smolAddCommandFromString(simptr sim,char *string);`
> Defines a runtime command, including its execution timing parameters, from the string `string`. This string should be identical to ones used in configuration files, except that they do not include the "cmd" statement.

Molecules

`enum ErrorCode smolAddSpecies(simptr sim,char *species,char *mollist);`
> Adds a molecular species named `species` to the system. If you have already created species lists and want all states of this species to live in a specific list, then enter it in `mollist`; otherwise, enter `mollist` as `NULL` or an empty string to request default behavior.

`int smolGetSpeciesIndex(simptr sim,char *species);`
> Returns the species index that corresponds to the species named `species`. Upon failure, this function returns an error code cast as an integer.

`char* smolGetSpeciesName(simptr sim,int speciesindex,char *species);`
> Returns the species name that corresponds to the species index in `speciesindex`. The name is returned both in `species` and directly, where the latter simplifies function use. Upon failure, this function returns `NULL`.

`enum ErrorCode smolSetSpeciesMobility(simptr sim,char *species,enum MolecState`
`state,double difc,double *drift,double *difmatrix);`

Sets any or all of the mobility coefficients for species `species` (which may be "all") and state `state` (which may be `MSall`). `difc` is the isotropic diffusion coefficient, `drift` is the drift vector, and `difmatrix` is the square of the anisotropic diffusion matrix (see the User's manual). To not set coefficients, enter a negative number in `difc` and/or enter a `NULL` pointer in the other inputs, respectively.

`int smolAddMolList(simptr sim,char *mollist);`
Adds a new molecule list, named `mollist`, to the system.

`int smolGetMolListIndex(simptr sim,char *mollist);`
Returns the list index that corresponds to the list named `mollist`.

`char* smolGetMolListName(simptr sim,int mollistindex,char *mollist);`
Returns the molecule list name that corresponds to the molecule list with index `mollistindex`. The name is returned both in `mollist` and directly. On error, this function NULL.

`enum ErrorCode smolSetMolList(simptr sim,char *species,enum MolecState state,char *mollist);`
Sets the molecule list for species `species` (which may be "all") and state `state` (which may be `MSall`) to molecule list `mollist`.

`enum ErrorCode  smolSetMaxMolecules(simptr sim,int maxmolecules);`
Sets the maximum number of molecules that can simultaneously exist in a system to `maxmolecules`. At present, this function needs to be called for a simulation to run, although it will become optional once dynamic molecule memory allocation has been written.

`enum ErrorCode smolAddSolutionMolecules(simptr sim,char *species,int number,double *lowposition,double *highposition);`
Adds `number` solution state molecules of species `species` to the system. They are randomly distributed within the box that has its opposite corners defined by `lowposition` and `highposition`. Any or all of these coordinates can equal each other to place the molecules along a plane or at a point. Enter `lowposition` and/or `highposition` as `NULL` to indicate that the respective corner is equal to that corner of the entire system volume.

`enum ErrorCode smolAddCompartmentMolecules(simptr sim,char *species,int number,char *compartment);`
Adds `number` solution state molecules of species `species` to the compartment `compartment`. Molecules are randomly distributed within the compartment.

`enum ErrorCode smolAddSurfaceMolecules(simptr sim,int speciesindex,enum MolecState state,int number,int surface,enum PanelShape panelshape,int panel,double *position);`
Adds `number` molecules of species `species` and state `state` to surface(s) in the system. It is permissible for `surface` to be "all", `panelshape` to be PSall, and/or `panel` to be "all". If you want molecules at a specific position, then you need to enter a specific surface, panel shape, and panel, and then enter the position in `position`.

`int smolGetMoleculeCount(simptr sim,char *species,enum MolecState state);`
Returns the total number of molecules in the system that have species `species` ("all" is permitted) and state `state` (`MSall` is permitted). Any error is returned as the error code cast as an integer.

`enum ErrorCode smolSetMoleculeStyle(simptr sim,const char *species,enum MolecState state,double size,double *color);`
Sets the graphical display parameters for molecules of species `species` ("all" is permitted) and state `state` (`MSall` is permitted). Enter `size` with the drawing size (in pixels if graphics method is "opengl" and in simulation system length units for better drawing methods) or with a negative number to not set the size. Enter `color` with the 3-value color vector or with `NULL` to not set the color.

Surfaces

`enum ErrorCode smolSetBoundaryType(simptr sim,int dimension,int highside,char type);`
>   Sets the molecule interaction properties for a system boundary that bounds the `dimension` axis. Enter
>   `dimension` as -1 to indicate all dimensions. Set `highside` to 0 for the lower boundary, to 1 for the upper
>   boundary, and to -1 for both boundaries. The boundary type is entered in `type` as 'r' for reflecting, 'p'
>   for periodic, 'a' for absorbing, or 't' for transmitting. Note that Smoldyn only observes these properties
>   if no surfaces are declared; otherwise all boundaries are transmitting regardless of what's entered here.

`int smolAddSurface(simptr sim,char *surface);`
>   Adds a surface called `surface` to the system.

`int smolGetSurfaceIndex(simptr sim,char *surface);`
>   Returns the surface index that corresponds to the surface named `surface`. The index is non-negative.
>   On failure, this returns an error code cast as an integer.

`char* smolGetSurfaceName(simptr sim,int surfaceindex,char *surface);`
>   Returns the surface name for surface number `surfaceindex` both directly and in the `surface` string.
>   On failure, this returns NULL.

`enum ErrorCode smolSetSurfaceAction(simptr sim,char *surface,enum PanelFace face,char *species,enum MolecState state,enum SrfAction action);`
>   Sets the action that should happen when a molecule of species `species` (may be "all") and state
>   `state` (may be `MSall`) diffuses into face `face` (may be `PFboth`) of surface `surface`. The action is set
>   to `action`.

`enum ErrorCode smolSetSurfaceRate(simptr sim,char *surface,char *species,enum MolecState state,enum MolecState state1,enum MolecState state2,double rate,char *newspecies,int isinternal);`
>   Sets the surface interaction rate(s) for surface `surface` (may be "all") and species `species` (may
>   be "all") and state `state`. The transition being considered is from `state1` to `state2` (this function
>   uses the tri-state format for describing surface interactions, shown below). The interaction rate is
>   set to `rate`, which is interpreted as a probability value for internal use if `isinternal` is 1 and as
>   a physical interaction coefficient if `isinternal` is 0. If the molecule ends up interacting with the
>   surface, it changes to new species `newspecies`. Enter `newspecies` as either NULL or an empty string
>   to indicate that molecules should not change species upon interactions. The molecule states are most
>   easily understood with the following table. If the action listed in the table is in italics, then the
>   corresponding combination of states is not a permitted input.

interaction class	tristate format			action
	state	state1	state2	
	soln	soln	soln	*reflect*
	"	"	bsoln	transmit
collision from	"	"	bound	adsorb
solution state	"	bsoln	soln	transmit
	"	"	bsoln	*reflect*
	"	"	bound	adsorb
	"	bound	soln	desorb
action from	"	"	bsoln	desorb
bound state	"	"	bound	*no change*
	"	"	bound'	flip
	bound	soln	soln	*reflect*
	"	"	bsoln	transmit
	"	"	bound	hop
collision from	"	"	bound'	hop
bound state	"	bsoln	soln	transmit

	”	”	bsoln	*reflect*
	”	”	bound	hop
	”	”	bound’	hop
	”	bound	soln	desorb
action from	”	”	bsoln	desorb
bound state	”	”	bound	*no change*
	”	”	bound’	flip
impossible	”	bound’	any	*nonsense*

int smolAddPanel(simptr sim,char *surface,enum PanelShape panelshape,char *panel,char *axisstring,double *params);
Adds or modifies a panel of shape `panelshape` of surface `surface`. `axisstring` 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. These are exactly the same parameters that are listed for the "panel" statement in Smoldyn configuration files, with the sole exception that the first rectangle "parameter" is actually a string that is entered in `axisstring`. `panelname` is an optional parameter for naming the panel; if it is included and is not an empty string, the panel is named `panelname`. 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 `panelname` as either NULL or as an empty string. In the latter case, `panelname` is returned with the newly assigned default name.

int smolGetPanelIndex(simptr sim,char *surface,enum PanelShape *panelshapeptr,char *panel);
Returns the panel index for the panel called `panel` on surface `surface`. If `panelshapeptr` is not NULL, this also returns the panel shape in `panelshapeptr`. On failure, this returns the error code cast as an integer.

char* smolGetPanelName(simptr sim,char *surface,enum PanelShape panelshape,int panelindex,char *panel);
Returns the name of the panel that is in surface `surface`, has shape `panelshape`, and has index `panelindex`, both directly and in the string `panel`. On failure, this returns NULL.

enum ErrorCode smolSetPanelJump(simptr sim,const char *surface,const char *panel1,enum PanelFace face1,const char *panel2,enum PanelFace face2,int isbidirectional);
Sets a jumping link between face `face1` of panel `panel1` and face `face2` of panel `panel2` of surface `surface`. The link goes from `panel1` to `panel2` if `bidirectional` is entered as 0 and goes in both directions if `bidirectional` is entered as 1. None of the surface, panel, or face entries is allowed to be "all". This does not set the actions of any species to "jump", which has to be done using the `smolSetSurfaceAction` function.

enum ErrorCode smolAddSurfaceUnboundedEmitter(simptr sim,const char *surface,enum PanelFace face,const char *species,double emitamount,double *emitposition);
Adds information about a point molecular source so that face `face` of surface `surface` can have its absorption properties calculated so that the molecular concentrations will become the same as they would be if the surface weren't there at all. The point molecular source emits molecules of species `species`, with a rate of `emitamount` and is at location `emitposition`. The emission rate does not need to be in absolute units, but only has to be correct relative to other unbounded emitters. None of the inputs to this function are allowed to be "all".

enum ErrorCode smolSetSurfaceSimParams(simptr sim,const char *parameter,double value);
Sets the surface simulation parameter named with `parameter` to value `value`. The possible parameters are "epsilon", "margin", and "neighbordist". In all cases, the defaults are nearly always good, although

this function allows them to be modified if desired. Epsilon is the maximum distance away from a surface that Smoldyn is allowed to place a surface-bound molecule. Margin is the distance inside from the edge of a surface panel that Smoldyn will place surface-bound molecules that hop onto this panel. Neighbor distance is the maximum distance over which surface-bound molecules are allowed to hop to transition from one panel to a neighboring panel.

`enum ErrorCode smolAddPanelNeighbor(simptr sim,const char *surface1,const char *panel1,const char *surface2,const char *panel2,int reciprocal);`
Adds panel `panel2` of surface `surface2` as a neighbor of panel `panel1` or surface `surface1`, meaning that surface-bound molecules will be allowed to diffuse from `panel1` to `panel2`. These are not allowed to be the same panel. Also, "all" values are not permitted. Otherwise, essentially any possible entries are legitimate. If surface-bound molecules should also be allowed to diffuse from `panel2` to `panel1`, enter `reciprocal` as 1; if not, enter `reciprocal` as 0.

`enum ErrorCode smolSetSurfaceStyle(simptr sim,const char *surface,enum PanelFace face,enum DrawMode mode,double thickness,double *color,int stipplefactor,int stipplepattern,double shininess);`
Sets the graphics output style for face `face` of surface `surface`. `mode` is the drawing mode; enter it as `DMnone` to not set this parameter and otherwise enter it as `DMno` to not draw the surface, `DMvert` for vertices, `DMedge` for edges, or `DMface` for faces. The `thickness` parameter gives the point size or line width for drawing vertices or edges, or can be entered as a negative number to not set this parameter. `color` is the 4-value color vector for the surface, or can be entered as `NULL` to not set this parameter. `stipplefactor` is the repeat distance for the entire edge stippling pattern, or can be entered as a negative number to not set it. `stipplepattern` is the edge stippling pattern, which needs to be between 0 and 0xFFFF, or can be entered as -1 to not set this parameter. And `shininess` is the surface shininess, for use with lighting in the "opengl_better" graphics display option, or can be entered as -1 to not set this parameter. The parameters `thickness`, `stipplefactor`, and `stipplepattern` only apply to edge style drawing modes and ignore any input in the `face` entry. The `shininess` parameter only applies to the face style drawing modes.

Compartments

`int smolAddCompartment(simptr sim,char *compartment);`
Adds a compartment called `compartment` to the system.

`int smolGetCompartmentIndex(simptr sim,char *compartment)`
Returns the index of the compartment named `compartment`. On failure, this returns an error code cast as an integer.

`char* smolGetCompartmentName(simptr sim,int compartmentindex,char *compartment)`
Returns the name of the compartment that has index `compartmentindex` both directly and in the string `compartment`. Returns `NULL` if an error arises.

`enum ErrorCode smolAddCompartmentSurface(simptr sim,char *compartment,char *surface);`
Adds surface `surface` as one of the bounding surfaces of compartment `compartment`.

`enum ErrorCode smolAddCompartmentPoint(simptr sim,char *compartment,double *point);`
Adds `point` as one of the interior-defining points of compartment `compartment`.

`enum ErrorCode smolAddCompartmentLogic(simptr sim,char *compartment,enum CmptLogic logic,char *compartment2);`
Modifies the current definition of compartment `compartment` using a logical rule specified in `logic` and the definition of `compartment2`.

Reactions

enum ErrorCode smolAddReaction(simptr sim,const char *reaction,const char *reactant1,enum MolecState rstate1,const char *reactant2,enum MolecState rstate2,int nproduct,const char **productspecies,enum MolecState *productstates,double rate);
Adds reaction named **reaction** to the system. This reaction can have up to two reactants, whose species are listed in **reactant1** and **reactant2** and whose states are listed in **rstate1** and **rstate2**. If the reaction has fewer than two reactants, set either or both of **reactant1** and **reactant2** to either **NULL** or an empty string. State the number of reaction products in **nproduct**, list their species in **productspecies**, and list their states in **productstates**. To set the reaction rate, enter it in **rate**; otherwise, enter **rate** as a negative number.

int smolGetReactionIndex(simptr sim,int *orderptr,char *reaction);
Returns the index and order for the reaction that is named **reaction**. If the order is known, send in **orderptr** pointing to this value. If it is not known, send in **orderptr** equal to either **NULL** or pointing to a negative number; in this case, it will be returned pointing to the reaction order, if the reaction was found. On failure, this returns the error code, cast as an integer.

char* smolGetReactionName(simptr sim,int order,int reactionindex,char *reaction);
Returns the name of the reaction that has reaction order **order** and index **reactionindex** in the string **reaction**. Also returns the result directly. Returns **NULL** if an error arises.

enum ErrorCode smolSetReactionRate(simptr sim,int order,char *reaction,double rate,int isinternal);
Set the reaction rate to **rate**. If this value is to be interpreted as an internal reaction rate parameter, meaning the production rate for zeroth order reactions, the reaction probability for first order reactions, or the binding radius for second order reactions, then set **isinternal** to 1. Rather than calling this function at all, it's usually easier to use the **rate** parameter of the **smolAddReaction** function, although that doesn't cope with internal rate values.

enum ErrorCode smolSetReactionRegion(simptr sim,const char *reaction,const char *compartment,const char *surface);
Limits the spatial region where a reaction can take place to the compartment **compartment** and/or the surface **surface**. To not set one of these limits, enter **compartment** and/or **surface** as NULL. To remove a previously set limit, enter **compartment** and/or **surface** as the empty string, "".

enum ErrorCode smolSetReactionProducts(simptr sim,const char *reaction,enum RevParam method,double parameter,const char *product,double *position);
Sets the reaction product parameters for reaction **reaction**. At a minimum, the **method** reversible parameter is required. Most of these methods require a single parameter, entered in **parameter**. A few methods also require a product, in **product** and the relative position of this product in **position**.

method	parameter	product	position
RPnone	-	-	-
RPirrev	-	-	-
RPconfspread	-	-	-
RPbounce	σ_u	-	-
RPpgem	ϕ	-	-
RPpgemmax	ϕ_{max}	-	-
RPpgemmaxw	ϕ_{max}	-	-
RPratio	σ_u/σ_b	-	-
RPunbindrad	σ_u	-	-
RPpgem2	ϕ	-	-
RPpgemmax2	ϕ_{max}	-	-
RPratio2	σ_u/σ_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.

Ports

`enum ErrorCode smolAddPort(simptr sim,const char *port,const char *surface,enum PanelFace face);`

Adds a port to the simulation. The port will be named `port` and will port at the `face` face of surface `surface`.

`int smolGetPortIndex(simptr sim,const char *port);`

Returns the index of the port named `port`.

`char* smolGetPortName(simptr sim,int portindex,char *port);`

Returns the name of the port with index `portindex`, both directly and in `port`.

`enum ErrorCode smolAddPortMolecules(simptr sim,const char *port,int nmolec,const char *species,double **positions);`

Adds `nmolec` molecules to Smoldyn's import buffer of port `port`. These molecules will all have species `species` and state `MSsoln`. Enter `positions` as `NULL` to have the molecules positioned randomly over the porting surface and as an `nmolec` length list of position vectors to have them located at those specific initial positions. These initial positions should be close to the porting surface, and on the Smoldyn system side of it.

`int smolGetPortMolecules(simptr sim,const char *port,const char *species,enum MolecState state,int remove);`

Returns the number of molecules that are in Smoldyn's export buffer of port `port`. Enter `species` with the species of the molecules that should be retrieved, or "all" for all species. Enter `state` with the states of the molecules that should be retrieved, or `MSall` for all states. Enter `remove` with 1 to remove molecules from the export buffer after they are retrieved or with 0 to leave them in the buffer. If an error arises, this returns the error code cast as an integer.

3.8   Libsmoldyn and SWIG

Before swigging, I need to test C libsmoldyn on my laptop.

I'm in the process of enabling Libsmoldyn access from Python and other high-level languages using the SWIG wrapper generator. The following are simply notes to myself that I will convert later to more formal instructions.

The first step of the build process, before anything else, is to run SWIG to create the wrapper code. I have not written a SWIG input file yet, but simply added a few lines to the top of Libsmoldyn.h, within the `#ifdef SWIG` section. In the source/Smoldyn directory, enter:

```
swig -python -o libsmoldyn_pywrap.c libsmoldyn.h
```

This creates the files libsmoldyn_pywrap.c and libsmoldyn.py. The former is a C code wrapper for Libsmoldyn, which can be called from Python, while the latter is Python code that defines the functions and that calls the C code wrapper.

Next, go to the trunk directory and run ./boostrap.sh if needed. Then (for my desktop computer) configure with:

```
CFLAGS="-I/sw/include/python2.6"./configure --enable-libsmoldyn --enable-swig
```

For my laptop, I use "-I /Developer/SDKs/-MacOSX10.5.sdk/System/Library/Frameworks/Python.framework/Versions/2.5/include/python2.5/". The CFLAGS include portion is there because Python.h is required and this is where my Python.h file is. Ideally, I'd add a line to the configure.ac file that searched for Python.h and included it automatically. Also, I'd like

to change the configure options to –enable-libsmoldyn for the C file, –enable-libsmoldyn-py for the Python bindings –enable-libsmoldyn-r for the R bindings, and so on. Multiple options could be used. Then run make and sudo make install.

If I enter `echo $PYTHONPATH`, I'm told that Python will look in /usr/local/lib/python2.6/site-packages/. So, I copied libsmoldyn.py and the libsmoldyn.0.dylib library into that directory, and also renamed the latter file to _libsmoldyn.so, using:

```
sudo cp source/Smoldyn/libsmoldyn.py /usr/local/lib/python2.6/site-packages/

sudo cp /usr/local/lib/libsmoldyn.0.dylib /usr/local/lib/python2.6/site-packages/
_libsmoldyn.so
```

Again, here are places where things could be improved. When the user types "sudo make install", the build system should automatically copy these files, with the correct names, into the correct places. At this point, I can call Libsmoldyn from Python. I start Python and enter `import libsmoldyn`. The most basic test that can be done next is to enter `print libsmoldyn.smolGetVersion()` and the version number should print out.

Other things to try include, for example, `libsmoldyn.smolSetDebugMode(1)`. I can also load files with `libsmoldyn.smolPrepareSimFromFile` and run them with `libsmoldyn.smolRunSim`. I haven't figured out how to do much more than this yet though.

Chapter 4

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. (5) Smoldyn also uses the Libmoleculizer module, which is C++ and Python code. It is fairly separate from the rest of the Smoldyn program and communicates through the smolmoleculizer.c interface functions.

4.1   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
smolmoleculizer.c	interface to Libmoleculizer, for reaction network generation
smolport.c	ports
smolreact.c	reactions
smolsim.c	simulation structure and high level functions
smolsurface.c	surfaces
smolwall.c	walls

4.2   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.

4.3   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.

4.4   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 5

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.

5.1 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. Following is the complete smoldyn.h header file.

```
/* Steven Andrews, started 10/22/2001.
 This is the header for the Smoldyn program.  See documentation
 called Smoldyn_doc1.pdf and Smoldyn_doc2.pdf.
 Copyright 2003-2011 by Steven Andrews.  This work is distributed under the terms
 of the Gnu General Public License (GPL). */

#ifndef __smoldyn_h__
#define __smoldyn_h__

#include <time.h>
```

```c
#include <stdio.h>

#if OPTION_VCELL
  #include <string>
  using namespace std;
#endif

/******************************* General *********************************/

#define DIMMAX 3              // maximum system dimensionality
#define VERYCLOSE 1.0e-12     // distance that's safe from round-off error

enum StructCond {SCinit,SClists,SCparams,SCok};

/****************************** Molecules ********************************/

#define MSMAX 5
#define MSMAX1 6
enum MolecState {MSsoln,MSfront,MSback,MSup,MSdown,MSbsoln,MSall,MSnone,MSsome};
enum MolListType {MLTsystem,MLTport,MLTnone};

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
  } *moleculeptr;

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
  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]
  } *molssptr;

/********************************* 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;

/******************************* Reactions *******************************/

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

#if OPTION_VCELL
class ValueProvider;
typedef ValueProvider* valueproviderptr;
#endif

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]
#if OPTION_VCELL
  valueproviderptr rateValueProvider;    // requested reaction rate
#endif
  double rate;                  // requested reaction rate
  double bindrad2;              // squared binding radius, if appropriate
  double prob;                  // reaction probability
  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]
  struct compartstruct *cmpt;   // compartment reaction occurs in, or NULL
  struct surfacestruct *srf;    // surface reaction on, or NULL
  } *rxnptr;

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;


/******************************** Surfaces ********************************/

#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};


typedef struct surfactionstruct {
  int *srfnewspec;              // surface convert mol. species [ms]
  double *srfrate;              // surface action rate [ms]
#if OPTION_VCELL
  valueproviderptr* srfRateValueProvider; //rate for surface actions: asorption,
      desorption, transmission...etc.
#endif
  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;

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 front[DIMMAX];         // 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;

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]
  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]
  } *surfaceptr;

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;

/********************************* Boxes *********************************/

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;

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;


/***************************** Compartments *****************************/

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;

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 [c]
  compartptr *cmptlist;       // list of compartments [c]
  } *compartssptr;


/*********************************** Ports ***********************************/

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;

/****************************** Filaments ********************************/

typedef struct filamentstruct {
  struct filamentsuperstruct *filss;  // filament superstructure
  char *fname;                  // filament 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
  int nmax;                     // number of monomers allocated [1,inf)
  int n;                        // number of monomers
  int front;                    // front index
  int back;                     // back index
  double **px;                  // Coords. for monomer ends [nmax+1][3]
  double *pl;                   // monomer length [nmax]
  double **pa;                  // relative ypr angles [nmax][3]
  double **pd;                  // relative dcm [nmax][9]
  double **po;                  // absolute monomer orientation [9]
  double *pthk;                 // thickness of monomer [nmax], [0,inf)
  double lstd;                  // minimum energy monomer length
  double astd[3];               // minimum energy bend angle
  double lk;                    // force constant for length
  double ak[3];                 // force constant for angle
  double kT;                    // thermodynamic temperature, [0,inf)
  double treadrate;             // treadmilling rate constant
  char surf;                    // character for surface shape
  double spar[2];               // parameters of surface
  } *filamentptr;

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;


/****************************** Moleculizer ********************************/

typedef struct mzrsuperstruct {
  enum StructCond condition;          // structure condition
  struct simstruct *sim;              // simulation structure
  struct moleculizer_handle *mzr;     // moleculizer object
  int ruleschars;                     // number of characters in rules
  char *rules;                        // input rules file
  int maxstreams;                     // allocated number of species streams
  int nstreams;                       // actual number of species streams
  char **streamname;                  // names of species streams [strm]
  double **displaysize;               // display sizes for streams [strm][ms]
  double ***color;                    // colors for streams [strm][ms][c]
  double **strmdifc;                  // diff. coeff. for streams [strm][ms]
  enum MolecState *strmstate;         // default states for streams [strm]
  int maxNetworkSpecies;              // maximum expansion size of network
  unsigned int maxnamehash;           // allocated size of name hash
```

```
  unsigned int nnamehash;           // actual size of name hash
  char **tagname;                   // hash list of mzr tagged names
  char **smolname;                  // hash list of Smoldyn names
  int maxrxnhash;                   // allocated size of reaction hash
  int nrxnhash;                     // actual size of reaction hash
  char **mzrrxn;                    // hash list of mzr reaction names
  char **smolrxn;                   // hash list of Smoldyn reaction names
  int maxspecies;                   // allocated size of species list
  enum MolecState *defaultstate;    // default state for each species [i]
  int refspecies;                   // species for diffusion coeff. reference
  double refmass;                   // mass of reference species
  double refdifc[MSMAX];            // diffusion coefficients of ref. species
  int expandall;                    // flag for full expansion at initialize
  } *mzrssptr;

/****************************** Threading ******************************/
//?? nothing in this section is documented yet
#define BASE_STACK_SIZE sizeof(int) * 16;

typedef struct stackstruct { // Used internally by threads for input/output
  void* stack_data;
  size_t current_size;
  size_t max_size;
  } stack;


typedef void* thread_t;


typedef struct threadstruct { // All the data needed to execute a thread.
  thread_t thread_id;
  stack* input_stack;
  stack* output_stack;
  } *threadptr;


typedef struct threadingsuperstruct { // Master structure that contains all the
    information for all threads.
  threadptr* thread;
  int nthreads;
  } *threadssptr;

/****************************** Graphics ******************************/

#define MAXLIGHTS 8            // must be ? GL_MAX_LIGHTS
enum LightParam {LPambient,LPdiffuse,LPspecular,LPposition,LPon,LPoff,LPauto,
    LPnone};

typedef struct graphicssuperstruct {
  enum StructCond condition;  // structure condition
  struct simstruct *sim;      // simulation structure
  int graphics;               // graphics: 0=none, 1=opengl, 2=good opengl
  int runmode;                // 0=Smoldyn, 1=Libsmoldyn
  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]
  enum LightParam roomstate;     // on, off, or auto (on)
  double ambiroom[4];            // global ambient light [c]
  enum LightParam lightstate[MAXLIGHTS];  // on, off, or auto (off) [lt]
  double ambilight[MAXLIGHTS][4];    // ambient light color [lt][c]
  double difflight[MAXLIGHTS][4];    // diffuse light color [lt][c]
  double speclight[MAXLIGHTS][4];    // specular light color [lt][c]
  double lightpos[MAXLIGHTS][3];     // light positions [lt][d]
  } *graphicsssptr;

/***************************** Simulation *******************************/

#define ETMAX 10
enum SmolStruct {SSmolec,SSwall,SSrxn,SSsurf,SSbox,SScmpt,SSport,SSfilament,SScmd,
    SSmzr,SSsim,SScheck,SSall,SSnone};
enum EventType {ETwall,ETsurf,ETdesorb,ETrxn0,ETrxn1,ETrxn2intra,ETrxn2inter,
    ETrxn2wrap,ETimport,ETexport};

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);

#if OPTION_VCELL
  struct CompartmentIdentifierPair {
    char name[128];
    unsigned char pixel;//the compartmentID
  };

  typedef struct VolumeSamples {
    int num[3];//number of mesh points in X, Y,Z
    double size[3];//actual size in X, Y, Z (e.g. in micron)
    double origin[3];//origin of the X, Y, Z
    unsigned char* volsamples;//compartmentID for each mesh point center
    int nCmptIDPair; // number of compartments
    CompartmentIdentifierPair* compartmentIDPairPtr;//ID vs. comptName pairs.
  }* VolumeSamplesPtr;

  class ValueProviderFactory;
  class AbstractMesh;
#endif

typedef struct simstruct {
  enum StructCond condition;   // structure condition
  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 tbreak;                // simulation break time
  double dt;                    // simulation time step
  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
  filamentssptr filss;          // filament superstructure
  mzrssptr mzrss;               // network generation rule superstructure
  void* cmds;                   // command superstructure
  graphicsssptr 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

#if OPTION_VCELL
  VolumeSamplesPtr volumeSamplesPtr;
  ValueProviderFactory* valueProviderFactory;
  AbstractMesh* mesh;
#endif
  } *simptr;


#if OPTION_VCELL
  class ValueProvider {
  public:
    virtual ~ValueProvider(){};
    virtual double getConstantValue()=0;
    virtual double getValue(double t, double x, double y, double z, rxnptr rxn)=0;
    virtual double getValue(double t, double x, double y, double z, rxnptr rxn,
        char* panelName)=0;
    virtual double getValue(double t, double x, double y, double z, surfactionptr
        actiondetails, char* panelName)=0;
  };

  class ValueProviderFactory {
  public:
    virtual ~ValueProviderFactory(){};
    virtual ValueProvider* createValueProvider(string& rateExp)=0;
    void setSimptr(simptr sim){this->sim = sim;}
    simptr getSimptr(){return this->sim;}
```

```
  private:
    simptr sim;
  };

  class AbstractMesh{
  public:
    virtual ~AbstractMesh(){};
    virtual void getCenterCoordinates(int volIndex, double* coords)=0;
    virtual void getDeltaXYZ(double* delta)=0;
    virtual void getNumXYZ(int* num)=0;
  };
#endif


#endif
```

5.2 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 {MLTsystem,MLTport,MLTnone};
```

 `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.

```
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
  } *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.

```
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
  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]
  } *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. 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 that is 1 if a species should be expanded with Libmoleculizer and 0 if not. Default is 0. Change it with the `molsetexpansionflag` function.

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 indicies. 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 indicies 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.

`int molwildcardname(molssptr mols,char *name,int channel,int itest);`
> Used for representing molecule species names using the wildcard characters '?' and '*'. This function has several modes. (1) Enter `mols` as NULL to free any allocated memory and reset all static variables to their default values. For all other modes, enter `mols` as the molecule superstructure. (2) Enter the species name, with wildcards, in `name` and enter either a 0 or a 1 in `channel` to set up a wildcard matching search for the given string. This clears any prior search in this channel. The function is called this way by `readmolname`. (3) Enter `name` as NULL and `itest` as zero or a negative number to return the index of the next species whose name matches the string in the listed channel, or -1 if there are no more matches. (4) Enter `name` as NULL and `itest` as a positive number to test whether species `itest` has a name that matches the string in the listed channel. Mode 2 can be combined with either mode 3 or mode 4; in the former case, the first match is returned, and then it is returned again on the next function call if mode 3 is used. Mode 4 works quickly with repeated calls because all results are pre-calculated and stored. Returns a species index, or -1 for no more matches in mode 3, -2 for failure to allocate memory, or -3 to -6 for syntax errors in the pattern.
>
> Here's an example. This function is first called with `name` equal to "A_*" and with `itest` equal to -1. The function returns 2 to indicate that species number 2 (say, "A_cyto") is a match to this string. On subsequent calls, now using mode 3, the function returns the species indices of species called "A_cyto", "A_nucleus", and "A_extracell" sequentially, and it would then return -1 to indicate no further matches. Alternatively, instead of those subsequent calls (or, after them), using mode 4, the function is called with the index of "A_nucleus" and it returns 1 and it is then called with the index of "B_cyto" and it returns 0.
>
> If the number of species changes during program operation, this function only updates its record of the number of species when a new species name is entered (mode 2). This is complicated though, so consult the code if it's important.

`int readmolname(simptr sim,char *str,enum MolecState *msptr,int channel);`
> Reads the first word of string `str` to parse the molecule species name and state. If `msptr` is not NULL, the state is returned in `msptr`; the species name is returned normally. The input format for the first

word is either just *species*, in which case `msptr` is returned pointing to `MSsoln`, or is *species(state)*, in which case `msptr` is returned pointing to whatever state is requested; "none" is not a permitted state, although all other options are permitted, including "all" and "bsoln". The species name can be "all" to represent all species or it can include wildcards (for which the `channel` input is used) as described below. There are many possible negative return values: -1 implies that the string cannot be read, -2 implies that the close parenthesis is missing or in the wrong place, -3 implies that the state cannot be read, -4 implies that the species name is unknown, -5 implies that the species name is "all", and -6 implies that the species name included a wildcard character and that the name matches at least one existing species name.

If you don't want this function to find matches that use wildcard characters, then enter `channel` as -1. Otherwise, enter `channel` as either 0 or 1, depending on which channel of `molwildcardname` that you want to use. If the species name includes wildcards, then this function returns -6 if at least one species name matches the input and -4 if no species name matches the input. In the former case, retrieve species indices by calling `molwildcardname`, using a `NULL` value for the `name` input and the same channel as before, sequentially until that function returns -1.

`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. `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 it is to be bound to a panel, 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. It is permissible for i to equal 0 for the molecule to be killed, which is equivalent to calling `molkill`.

`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,enum MolecState ms,double difc);`
Sets the diffusion coefficient for molecule `ident` and state `ms` to `difc`. Sets the diffusion coefficient for all species if `ident` is -5 and for wildcard selected species (channel 0) if `ident` is -6. If `ms` is `MSall`, this sets the diffusion coefficient for all states. Either or both "all" conditions are permitted. This does not update rms step sizes or reaction rates.

`int molsetdifm(simptr sim,int ident,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. Sets the diffusion matrix for all species if `ident` is -5 and for wildcard selected species (channel 0) if `ident` is -6. If `ms` is `MSall`, this sets the diffusion matrix for all states. Either or both "all" conditions are permitted. 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,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. Sets the drift vector for all species if `ident` is -5 and for wildcard selected species (channel 0) if `ident` is -6. If `ms` is `MSall`, this sets the drift vector for all states. Either or both "all" conditions are permitted. This returns 0 for successful operation and 1 for failure to allocate memory.

`int molsetsurfdrift(simptr sim,int ident,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. Sets the surface drift vector for all species if `ident` is -5 and for wildcard selected species (channel 0) if `ident` is -6. 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,enum MolecState ms,double dsize);`
> Sets the display size for molecule `ident` and state `ms` to `dsize`. Sets the display size for all species if `ident` is -5 and for wildcard selected species (channel 0) if `ident` is -6. If `ms` is `MSall`, this sets the display size for all states. Either or both "all" conditions are permitted.

`void molsetcolor(simptr sim,int ident,enum MolecState ms,double *color);`
> Sets the color for molecule `ident` and state `ms` to the 3-dimensional RGB vector `color`. Sets the color for all species if `ident` is -5 and for wildcard selected species (channel 0) if `ident` is -6. If `ms` is `MSall`, this sets the color for all states. Either or both "all" conditions are permitted.

`void molsetlistlookup(simptr sim,int ident,enum MolecState ms,int ll);`
> Sets the list lookup table value to live list number `ll` for molecule `ident` and state `ms`. Various special codes are possible: `ident=-5` implies all identities, `ident=-6` implies wildcard selected species (channel 0), `ident=-7` implies all diffusing molecules, and `ident=-8` implies all non-diffusing molecules; `ms=MSall` implies all states.

`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,boxptr bptr,int max);`
> Counts the number of molecules of type `i` and state `ms` currently in the simulation. If `bptr` is NULL, this considers all molecules in the simulation, and otherwise it only considers those in the listed box. 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 -5 and enter `ms` as `MSall`. All system type molecule lists and the dead list are checked, meaning that any porting lists are not included. If `bptr` is NULL, 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. Also, `i` can be -6 to indicate those species that are selected with a wildcard character, using channel 0.

`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.

`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 indicies.

`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, -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 molsetexpansionflag(simptr sim,int i,int flag);
```
> Sets the expansion flag for species `i` to `flag`. Enter `i` as -1 for all species. A value of zero means that the species should not be expanded, non-zero menas that moleculizer should expand it. Returns 0 for success, 2 for a non-existant molecule superstructure, or 3 if `i` is out of bounds.

```
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. 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);
```
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 indicies. 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.

`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).

5.3   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.

`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.

<u>core simulation functions</u>

```
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.

5.4   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. 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.

```
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]
  double rate;                  // requested reaction rate
  double bindrad2;              // squared binding radius, if appropriate
  double prob;                  // reaction probability
  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]
  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). `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 $molecules * volume^{(\mathtt{order}-1)}/time$. The precise meaning of `rate` depends on the order of the reaction. `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. `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. `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.

```
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 `maxlistôrder` 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.

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

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".

low level utilities

`int readrxnname(simptr sim,char *rname,int *orderptr,rxnptr *rxnpt);`
> Using a reaction name in `rname`, this looks it up in the current list of reaction names, working with increasing reaction orders, until it finds it. If it is found, it returns the reaction order at `orderptr`, unless `orderptr` is NULL, and it returns a pointer to the reaction at rxnpt, unless rxnpt is NULL, and it returns the reaction number directly. If it is not found, it returns -1.

`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 singe states 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; 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, or 4 for non-confspread bimolecular reactions that have non-diffusing reactants.
>
> 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 probability of a unimolecular reaction occurring for an individual reactant molecule during one time step, which is $rate/sum \cdot [1 - exp(-sum \cdot dt)]$, where sum is the sum of the defined rate values for all unimolecular reactions of the reactant. 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`).

`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, it is assumed that the product position vectors and the unbinding radius have all values equal to 0 initially. 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`.

`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 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, RPratio, RPoffset, or RPnone, and not otherwise. 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. Returns 0 for success, 1 for missing input item, 2 for unknown option, 3 for a value that was set previously, 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 "allorad", the reaction 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.

`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`. 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	σ_u	-	-
RPpgem	ϕ	-	-
RPpgemmax	ϕ_{max}	-	-
RPpgemmaxw	ϕ_{max}	-	-

RPratio	σ_u/σ_b	-	-
RPunbindrad	σ_u	-	-
RPpgem2	ϕ	-	-
RPpgemmax2	ϕ_{max}	-	-
RPratio2	σ_u/σ_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 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.

`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 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.

core simulation functions

`int doreact(simptr sim,rxnptr rxn,moleculeptr mptr1,moleculeptr mptr2,int ll1,int m1,int ll2,int m2,double *pos,panelptr pnl);`

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). Reactants are killed, but left in the live lists. Any products are created on the dead list, for transfer to the 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. The `pos` and `pnl` 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.

`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.

5.5   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` = 2	`npts` = 4
`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
triangles, ps = PStri		
`npts` = 1	`npts` = 2	`npts` = 3
`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.

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

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	
	soln	front	fsoln	reflect
	"	"	bsoln	transmit
collision from	"	"	bound	adsorb
solution state	"	back	fsoln	transmit
	"	"	bsoln	reflect
	"	"	bound	adsorb
impossible	"	none	any	
	bound	front	fsoln	reflect
	"	"	bsoln	transmit
	"	"	bound	hop
collision from	"	"	bound'	hop
bound state	"	back	fsoln	transmit
	"	"	bsoln	reflect
	"	"	bound	hop
	"	"	bound'	hop
	"	none	fsoln	desorb
action from	"	"	bsoln	desorb
bound state	"	"	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.

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 Δt 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 front[DIMMAX];         // 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.

```
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]
  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]
  } *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.

   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.

```
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. `snames` is a list of names for the surfaces. `srflist` is the list of pointers to surfaces. `neighdist` is a poorly named element, which is used for the diffusion of molecules on a surface; if a surface-bound molecule diffuses off of a panel and it is within `neighdist` distance of a neighboring panel, then it moves to the neighboring panel. `neighdist` is initialized to -1 and later re-initialized to a value that depends on molecule diffusion coefficients. `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

```
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
collision from solution state	soln	front	fsoln	reflect	soln	front	fsoln
	”	”	bsoln	transmit	soln	back	fsoln
	”	”	bound	bind	bound	none	fsoln
	”	back	fsoln	transmit	soln	front	bsoln
	”	”	bsoln	reflect	soln	back	bsoln
	”	”	bound	bind	bound	none	bsoln
impossible	”	none	any	impossible	none	none	none
collision from bound state	bound	front	fsoln	reflect	bound	front	fsoln
	”	”	bsoln	transmit	bound	back	fsoln
	”	”	bound’	hop	*bound’*	*both*	*bound*
	”	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
	"	"	bound'	hop	bound	front	bound'
collision from	"	bsoln	soln	transmit	bound	back	fsoln
bound state	"	"	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.

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).

`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 (≤ 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 (≤ 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 (≤ 0).

`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,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 -6 to indicate wildcard selected species (channel 0). `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,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.

`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 3 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);`
> 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.

`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.

`int lineXpanel(double *pt1,double *pt2,panelptr pnl,int dim,double *crsspt,enum PanelFace *face1ptr,enum PanelFace *face2ptr,double *crossptr,double *cross2ptr,int *veryclose);`
> 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, `crsspt` 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.
>
> 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 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 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).

    The internal vector `norm` is the local surface normal, pointing towards the front side.

`void movept2panel(double *pt,panelptr pnl,int dim,double margin);`
    This moves the point `pt` so that it is over the panel `pnl`, and 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.

`double closestpanelpt(panelptr pnl,int dim,double *testpt,double *pnlpt);`
    Finds the closest point that is on panel `pnl` to the test point `testpt` and returns it in `pnlpt`. This also returns the distance between `testpt` and `pnlpt`, which is always positive.

`double closestsurfacept(surfaceptr srf,int dim,double *testpt,double *pnlpt,panelptr *pnlptr);`

Finds the closts point that is on the surface `srf` to the test point `testpt` and returns it in `pnlpt`. 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.

`void movemol2closepanel(simptr sim,moleculeptr mptr,int dim,double epsilon,double neighdist,double margin);`

Checks to see if 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 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`.

`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 two 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, and (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`. Quite possibly, the code would be better if these two uses were separated into two 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 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 a 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.

pthreads code

`void* check_surfaces_on_subset_mols(void* data);` This code has not been proofread or thoroughly tested.

`int checksurfaces_threaded( simptr sim, int ll, int reborn);` This code has not been proofread or thoroughly tested.

5.6   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`. This function should only be called if `mptr` is known to be listed in list `ll` of the box. Before returning, `mptr->box` is set to NULL.

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. 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.

5.7 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);`
> 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.

`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.

5.8   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);`

> 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. 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.

5.9 Filaments (functions in smolfilament.c)

I'm just starting to develop filament support. Thus, a few things work and most don't. At present, the code is a combination of a Smoldyn feature template that I largely cut and pasted from smolports.c, and functions that I cut and paste from my Polymers.c library. Those functions don't appear to have ever been run, but were nominally improved versions of those in my randomwalk.c library. Everything needs to be filled in, cleaned up, etc. In particular, structure member names are not particularly clear.

Data structures declared in smoldyn.h

```
typedef struct filamentstruct {
  struct filamentsuperstruct *filss;   // filament superstructure
  char *fname;                         // filament 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
  int nmax;                            // number of monomers allocated [1,inf)
  int n;                               // number of monomers
  double **px;                         // Coords. for monomer ends [nmax+1][3]
  double *pl;                          // monomer length [nmax]
  double **pa;                         // relative ypr angles [nmax][3]
  double **pd;                         // relative dcm [nmax][9]
  double **po;                         // absolute monomer orientation [9]
  double *pthk;                        // thickness of monomer [nmax], [0,inf)
  double lstd;                         // minimum energy monomer length
  double astd[3];                      // minimum energy bend angle
  double lk;                           // force constant for length
  double ak[3];                        // force constant for angle
  double kT;                           // thermodynamic temperature, [0,inf)
  double treadrate;                    // treadmilling rate constant
  char surf;                           // character for surface shape
  double spar[2];                      // parameters of surface
  } *filamentptr;

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;
```

  Text about structure.
Filament math

Following are the basic equations for the filament relative angles ($\mathbf{A}$, `pd`), absolute angles ($\mathbf{B}$, `po`), and positions ($\mathbf{x}$, `px`).

$$\mathbf{B}_0 = \mathbf{A}_0$$

$$\mathbf{B}_i = \mathbf{A}_i \cdot \mathbf{B}_{i-1}$$

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

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.

<u>low level utilities</u>

```
double filRandomLength(filamentptr fil,double thickness);
```
    Local. Returns a random monomer length for a filament.

```
int filAddMonomer(filamentptr fil,double *x,double length,double *angle,double
    thickness,char endchar);
```
    Adds a monomer to filament `fil`. If this is the first monomer, then `x` needs to be set to the starting
    location of the filament; otherwise, `x` is ignored. `length` is the length of the monomer, `angle` is the
    relative angle of the monomer facing along the polymer from front to back (for the first monomer,
    this is the absolute angle), `thickness` is the monomer 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 monomers 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 monomer is added to the back, then `angle` is the angle of the new monomer off of
    the coordinate system. If the first monomer is added to the front, then `angle` is the angle from the
    new monomer to the coordinate system.

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

    For monomers added to the back, with index i: $\mathbf{a}_i = $ `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 monomers added to the front, with index i (typically equal to 0):

    $\mathbf{a}_i = $ `angle`, $\mathbf{A}_i = DCM(\mathbf{a}_i)$, $\mathbf{B}_i = \mathbf{A}_i \cdot \mathbf{B}_{i-1}$ $\mathbf{x}_{i+1} = \mathbf{x}_i + l_i\mathbf{B}_i^T \cdot \hat{\mathbf{x}}$

5.10   Libmoleculizer (functions in smolmoleculizer.c)

Libmoleculizer is a library for automatic rule-based reaction network generation. It generates species and
their reactions based on rules that specify binding and modification sites on a core set of "mols". These
rules get loaded in with rest of the Smoldyn configuration file, but in a special rules section.

    Smoldyn compiles without Libmoleculizer by default. If desired, this can be over-ridden by configuring
with the `--enable-Libmoleculizer` option, as described above in chapter 2, and then making and installing
as normal. If Libmoleculizer is enabled, the pre-processor macro `LIBMOLECULIZER` will be defined.

    Smoldyn communicates with Libmoleculizer exclusively through the smolmoleculizer.c file. It is a
wrapper for Libmoleculizer and it does other things as well. Nathan Addy wrote the original version of
smolmoleculizer.c. I substantially rewrote it in January and February of 2010 to simplify the data structures,
clean up the code, and complete the functionality of several functions. Documentation for the deprecated
code was included with the Smoldyn 2.18 release but not thereafter.

    Most communication between Smoldyn and Libmoleculizer is through text names. In particular, every
species that Libmoleculizer knows about has a "tagged name", which is generated by Libmoleculizer. This
is up to 15 characters long and is not human-readable. These species also have "explicit names", which are
human readable names that the user assigns in the rules file and which correspond to Smoldyn names. These
explicit names are typically referred to in the smolmoleculizer.c file as smolnames.

    Another type of name are species stream names. The user declares these in a section of the rules file titled
explicit-species-class. These names are also used in a few Smoldyn statements where display parameters are
set for species streams.

<u>Data structures declared in smoldyn.h</u>

```
typedef struct mzrsuperstruct {
  enum StructCond condition;        // structure condition
  struct simstruct *sim;            // simulation structure
  struct moleculizer_handle *mzr; // moleculizer object
  int ruleschars;                   // number of characters in rules
  char *rules;                       // input rules file
  int maxstreams;                   // allocated number of species streams
  int nstreams;                      // actual number of species streams
  char **streamname;                 // names of species streams [strm]
  double **displaysize;             // display sizes for streams [strm][ms]
  double ***color;                   // colors for streams [strm][ms][c]
  double **strmdifc;                 // diff. coeff. for streams [strm][ms]
  int maxNetworkSpecies;            // maximum expansion size of network
  int maxnamehash;                   // allocated size of name hash
  int nnamehash;                     // actual size of name hash
  char **tagname;                   // hash list of mzr tagged names
  char **smolname;                   // hash list of Smoldyn names
  int maxrxnhash;                   // allocated size of reaction hash
  int nrxnhash;                      // actual size of reaction hash
  char **mzrrxn;                     // hash list of mzr reaction names
  char **smolrxn;                    // hash list of Smoldyn reaction names
  int maxspecies;                   // allocated size of species list
  enum MolecState *defaultstate;   // default state for each species [i]
  int refspecies;                   // species for diffusion coeff. reference
  double refmass;                   // mass of reference species
  double refdifc[MSMAX];            // diffusion coefficients of ref. species
  int expandall;                     // flag for full expansion at initialize
  } *mzrssptr;
```

As usual, `condition` tells about whether this data structure is up-to-date or not. `sim` points to the simulation structure. `mzr` points to the moleculizer object. The moleculizer object has its own memory and does all of the Libmoleculizer work. It is defined in libmzr_c_interface.h.

`rules` is the entire text of the rules file, which is allocated to `ruleschars` characters.

The moleculizer superstructure keeps track of the display parameters for each of many species streams. A total of `maxstreams` are allocated and `nstreams` of them are defined. Each stream has a name (`STRCHAR` characters) in `streamname`. The molecule display sizes for the streams and their states are listed in `displaysize` and the molecule colors are listed in `color`. Also, diffusion coefficients for new species can be fixed to values for the species stream. This is done with `strmdifc` if so. `strmdifc` is set to -1 by default, which means that it has not been set, while values ≥ 0 are values that will be copied to new species.

The reaction network should be expanded up to size `maxNetworkSpecies`, although this functionality has not been implemented yet. Expansion should be unlimited if `maxNetworkSpecies` is negative and limited otherwise.

The name hash converts between mzr tagged names and explicitly declared Smoldyn names. It is allocated to size `maxnamehash` and has `nnamehash` name pairs. The mzr tagged names are in `tagname` and the explicitly declared Smoldyn names are in `smolname`.

The reaction hash converts between mzr reaction names (human readable but very verbose) and Smoldyn names (basically just numbers). It is allocated to size `maxrxnhash` and has `nrxnhash` reaction name pairs. These name pairs are stored in `mzrrxn` and `smolrxn`.

So that new species are created with the correct states, the moleculizer superstructure can keep track of the default state for each Smoldyn species in the `defaultstate` list. This list is allocated to size `maxspecies` and has `sim->mols->nspecies` elements used. The `maxspecies` value should agree with the corresponding value in the molecule superstructure. The `defaultstate` is created on demand, so that this array is not created if it is not requested.

`refspecies`, `refmass`, and `refdifc` list the species index, its mass, and its diffusion coefficients in each state for a species that is serves as a reference for calculating the diffusion coefficients of newly generated species. Both the mass and the diffusion coefficients are copied over from other information sources.

`expandall` is a flag that is 0 for on-the-fly reaction network generation and 1 for full network expansion

upon initialization.

During simulation, there are very few links from core Smoldyn to Smolmoleculizer. When Smoldyn performs a reaction, it checks that each reaction product has already been expanded. If not, it calls `mzrExpandSpecies`. This function, in Smolmoleculizer, tells Libmoleculizer to expand the species. Then, if the reaction network grew, it downgrades the mzr superstructure condition. Then it returns. Later on, the core Smoldyn simulation notices that the mzr superstructure condition has been downgraded, so it calls `mzrsetupmoleculizer`. This function retrieves the new species and reactions from Libmoleculizer, adds them to Smoldyn, upgrades the mzr superstructure condition, and returns.

Additional note on Libmoleculizer (lmzr). In addition to the configuration file sections that lmzr parses as described here, it also parses sections titled "Parameters" and "Reaction-Rules". As near as I can tell, the Parameters section can be used to assign values to variables and to do basic calculations. It does not appear useful.

Function declarations. As much as possible, the smolmoleculizer 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.

low level utilities

`int mzrTagName2SmolName(simptr sim,char *tagname);`
> Local. This function takes in a name in `tagname`, which is either the name of a moleculizer species (a tagged name) or the name of a regular Smoldyn species (as opposed to just those Smoldyn species that were explicitly declared in the rules file). Either way, it returns the index of the Smoldyn species. It searches the moleculizer names first (using the `tagname` element) and then the Smoldyn names (`sim->mols->spname`). Possible errors: -1 if `tagname` is in neither list and -2 means `tagname` is a moleculizer name but the moleculizer name hash stored value for the corresponding Smoldyn name doesn't match any name in Smoldyn's records.

`int mzrSmolName2TagName(mzrssptr mzrss,char *smolname,char *tagname);`
> Local. This function is similar to `mzrTagName2SmolName`, in that it converts between the two names, but this function uses a different data source. This function uses the moleculizer object data, and is simply a wrapper for the Libmoleculizer function `convertSomeNameToTaggedName`. Enter a Smoldyn name in `smolname` and this function will look for a corresponding moleculizer tagged name and will return it in `tagname`. `tagname` should be pre-allocated to size `STRCHAR`. Returns 0 for success, 1 for an unknown error, 2 if no match was found, 3 if the tagged name is more than `STRCHAR` characters (which should never happen), or 4 if Libmoleculizer is not in the build.

`int mzrReadStreamName(char *str,char *streamname,enum MolecState *msptr);`
> Global. This function reads a species stream name, with an optional molecular state, from the input string `str`. It does not modify `str`. If the state is not given, `MSsoln` is assumed. The stream name is copied into the string `streamname` if `streamname` is not NULL; it should be pre-allocated to size `STRCHAR`. The state is returned in `msptr` if `msptr` is not NULL. This function is to be used for reading input in the `speciesstream color` and `display` statements in Smoldyn input files. No check is made on the species stream name regarding whether it is in the moleculizer object or not (or whether it has been previously declared, at all). Returns 0 for success, -1 if the string cannot be read, -2 if the close parenthesis is missing or in the wrong place, -3 if the state cannot be read, or -5 if the stream name is "all". This function is a slightly modified copy of `readmolname` in smolmolec.c.

`int mzrGetSpeciesStreams(mzrssptr mzrss,char ***streamnames,int *numnames);`
> Global. Gets the list of species stream names that have been declared in the rules file and are in the moleculizer object. Pass in pointers to variables in `streamnames` and in `numnames` that will be returned with the data; the former will be returned with a newly allocated array of stream names and the latter will be returned with the number of names. To free the memory allocated here, call `mzrFreeSpeciesStreams`. This function is a wrapper for Libmoleculizer's function

`getAllSpeciesStreams`. Returns 0 for success, 1 if memory could not be allocated, and 2 if Libmoleculizer is not in the build.

`void mzrFreeSpeciesStreams(char **streamnames,int numnames);`
Global. Frees species streams that became allocated through a call to `mzrGetSpeciesStreams`.

`int mzrIsTagNameInStream(mzrssptr mzrss,char *tagname,char *stream);`
Local. Checks to see if the tagged name `tagname` is in the species stream called `stream`. This uses the moleculizer object records and is a wrapper for the Libmoleculizer function `checkSpeciesTagIsInSpeciesStream`. Returns 0 if not, 1 if it is, and -1 for either an unknown failure or if Libmoleculizer is not in this build.

`int mzrIsSmolNameInStream(mzrssptr mzrss,char *smolname,char *stream);`
Global. Checks to see if the Smoldyn name `smolname` is in the species stream called `stream`. This uses the moleculizer object records. Returns 0 if not, 1 if it is, and -1 for either an unknown failure or if Libmoleculizer is not in this build.

`int mzrNumberOfSpecies(mzrssptr mzrss);`
Global. This function returns the number of species that moleculizer has recorded thus far. It is a wrapper for the Libmoleculizer `getNumberOfSpecies` function. Returns the number of species or 0 if Libmoleculizer is not available or has not been set up; also returns -1 for unknown error.

`int mzrNumberOfReactions(mzrssptr mzrss);`
Global. This function returns the number of reactions that moleculizer has recorded thus far. It is essentially a wrapper for the Libmoleculizer `getNumberOfReactions` function. Returns the number of reactions or 0 if Libmoleculizer is not available or if it has not been set up; also returns -1 for unknown error.

`int mzrGetSpeciesSymmetry(mzrssptr mzrss,char *smolname);`
Returns the symmetry of the species named `smolname`, or -1 if an error occurred. This is a wrapper for the Libmoleculizer function `getSpeciesSymmetryDegree`.

Memory management functions

`int mzrallocrules(mzrssptr mzrss,int ruleschars);`
Local. Allocates the `rules` element of the moleculizer superstructure to `ruleschars`. If the `rules` element already existed, this enlarges or shrinks it as necessary and keeps any prior data. Returns 0 for success, 1 for inability to allocate memory, or 2 for illegal inputs.

`int mzrallocstreams(mzrssptr mzrss,int maxstreams);`
Local. Allocates the `streamname`, `displaysize`, and `color` elements of the moleculizer superstructure for up to `maxstreams` streams, and updates the `maxstreams` element. If these arrays were already allocated, then this function copies any existing data over into new memory and frees the old memory. Returns 0 for success, 1 for inability to allocate memory, or 2 for illegal inputs (`NULL` `mzrss` or `maxstreams` <0).

`void mzrfreestreams(char **streamname,double **displaysize,double ***color,double **strmdifc,int maxstreams);`
Local. Frees `streamname`, `displaysize`, `color`, `strmdifc`, and all of their elements. Each entry should be allocated for `maxstreams` streams if allocated at all; partial setup is fine.

`int mzrallocnamehash(mzrssptr mzrss,int maxnamehash);`
Local. Allocates the `tagname` and `smolname` elements of the moleculizer superstructure for up to `maxnamehash` names, and updates the `maxnamehash` element. If these arrays were already allocated, then this function copies any existing data over into new memory and frees the old memory. Returns 0 for success, 1 for inability to allocate memory, or 2 for illegal inputs (`NULL` `mzrss` or `maxnamehash` $<$ 0).

`void mzrfreenamehash(char **tagname,char **smolname,int maxnamehash);`

Local. Frees `tagname` and `smolname` and all of their elements. Each entry should be allocated for `maxnamehash` names if allocated at all; partial setup is fine.

`int mzrallocrxnhash(mzrssptr mzrss,int maxrxnhash);`

Local. Allocates the `mzrrxn` and `smolrxn` elements of the moleculizer superstructure for up to `maxrxnhash` names, and updates the `maxrxnhash` element. If these arrays were already allocated, then this function copies any existing data over into new memory and frees the old memory. Returns 0 for success, 1 for inability to allocate memory, or 2 for illegal inputs (`NULL` `mzrss` or `maxnamehash` < 0). This is essentially identical to `mzrallocnamehash`.

`void mzrfreerxnhash(char **mzrrxn,char **smolrxn,int maxrxnhash);`

Local. Frees `mzrrxn` and `smolrxn` and all of their elements. Each entry should be allocated for `maxrxnhash` names if allocated at all; partial setup is fine. This is essentially identical to `mzrfreenamehash`.

`int mzrallocdefaultstate(mzrssptr mzrss,int maxspecies);`

Local. Allocates the `defaultstate` element of the moleculizer superstructure to a size of `maxspecies` and updates the `maxspecies` element. If `defaultstate` was already defined, this copies over the existing data to the new memory and frees the old memory. Returns 0 for success, 1 for inability to allocate memory, and 2 for illegal inputs.

`mzrssptr mzrssalloc(void);`

Local. Allocates a completely empty moleculizer superstructure. All elements are set to either 0 or `NULL`, as appropriate. Returns `NULL` if memory could not be allocated.

`void mzrssfree(mzrssptr mzrss);`

Global. Frees a Moleculizer superstructure, including all components in it.

Data structure output

`int mzrCheckParams(simptr sim,int *warnptr);`

Global. Checks a Moleculizer superstructure, returning the number of errors if any. If `warnptr` is not `NULL`, it is returned pointing to the number of warnings. Any errors or warnings are printed to stdout. This checks the basic structure set up. More tests could be added, but the current ones are probably adequate. It would be nice to check things within the moleculizer object too, but that may not be possible.

`void mzroutput(mzrssptr mzrss);`

Local. Prints various things out about the moleculizer model and rules, from the moleculizer object. This is called by `mzrssoutput`, so should not need to be called directly.

`void mzrssoutput(simptr sim);`

Global. Displays all parameters about the moleculizer superstructure. This calls `mzroutput` to display things about the moleculizer object. This does not display anything if no rules file has been loaded.

`void mzrsswrite (simptr sim,FILE *fptr);`

Global. Writes the moleculizer superstructures parameters to `fptr` in a way that they can be reloaded and run later on. *?? This function needs lots of work.*

Structure set up

`void mzrsetcondition(mzrssptr mzrss,enum StructCond cond,int upgrade);`

Local. This function sets the condition of the moleculizer 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 mzrAssignDrawingParameters(simptr sim,char *tagname,int ident);`

Local. Sets the color and display size for the moleculizer species tag in `tagname` and corresponding Smoldyn species number in `ident`. This is based on the species stream that `tagname` is a member of,

it any. Returns 0 for success, 1 if `tagname` was not found in any of Libmoleculizer's species streams, or 2 if Libmoleculizer is not defined or not sufficiently set up.

`int mzrAssignDiffCoeff(simptr sim,char *tagname,double mass,int ident);`
Local. Sets the diffusion coefficients for the moleculizer species tag in `tagname` and corresponding Smoldyn species number in `ident`. This new species has mass `mass`. This assigns the diffusion coefficient according to the following sequence: it uses the species stream diffusion coefficient if one is entered, the reference mass and diffusion coefficient if one is entered, or 0 if neither of the above.

`int mzrSetDefaultState(simptr sim,int ident,enum MolecState ms);`
Global. Sets the default state to `ms` for species `ident`. Allocates the `defaultstate` element and sets `maxspecies` if required. Returns 0 for success, 1 for out of memory, or 2 for illegal inputs.

`enum MolecState mzrMolState(mzrssptr mzrss,int ident,int molint);`
Local. This function returns the default state, if one is listed, for the `molint`'th mol that is listed in species number `ident`. This uses the text name that is listed for species `ident` (using `sim->mols->spname`) and parses it to find `molint`'th mol name. Then it looks for that name in the species list and sees if a default state is listed for it. Returns `MSnone` if it can't find any useful information and the state otherwise.

`enum MolecState mzrDetermineProductState(mzrssptr mzrss,int ident,enum MolecState ms1,enum MolecState ms2);`
Local. This function determines the state of the reaction product species `ident`, based on optional reactant states `ms1` and `ms2`. It runs several tests. First, it sees if a default state has been stored for this particular species. If not, this sees if this species is part of a species class and if a state has been listed for that class. If there's a result but it's not consistent with the reactant states (because the product is surface-bound and the reactants aren't), then the result is set back to `MSnone` to indicate that it's invalid. Finally, if there's still no result, this makes its best guess based on the reactant states. At the end, this function returns its best guess for the product state, and returns `MSnone` if there is absolutely no information to work from.

`int mzrAssignProductDefaultState(mzrssptr mzrss,int *reactants,int nreactants,int *products,int nproducts);`
Local. This function determines the default state of reaction product species by calling mzrDetermineProductState. `reactants` is a list of `nreactants` Smoldyn species indices and `products` is a list of `nproducts` Smoldyn species indices. Returns 0 for success, 1 for memory error, or 2 for illegal parameters.

`int mzrSetStreamDisplay(mzrssptr mzrss,char *streamname,enum MolecState ms,double displaysize,double *color);`
Global. Sets the display size and/or color for the species stream called `streamname` and state `ms` to `displaysize` and `color`, respectively. To not set the display size, enter `displaysize` with a negative number and to not set the color enter `color` as `NULL`. If the stream name does not already exist, it is added to the list; the allocated list size is enlarged if necessary. The state, `ms`, can be `MSall` to set the display parameters for all states. Returns 0 for success, or 1 for out of memory.

`int mzrSetStreamDifc(mzrssptr mzrss,char *streamname,enum MolecState ms,double difc);`
Global. Sets the stream diffusion coefficient for the species stream called `streamname` and state `ms` to `difc`. Entering a negative number for `difc` will be recorded in the data structure, and indicates that other methods should be used for calculating diffusion coefficients for new species. If the stream name does not already exist, it is added to the list; the allocated list size is enlarged if necessary. The state, `ms`, can be `MSall` to set the display parameters for all states. Returns 0 for success, or 1 for out of memory.

`int mzrSetStreamState(mzrssptr mzrss,char *streamname,enum MolecState ms);`
Global. Sets the stream state for the species stream called `streamname` to `ms`. If the stream name does not already exist, it is added to the list; the allocated list size is enlarged if necessary. Returns 0 for success, or 1 for out of memory.

`int mzraddtonamehash(mzrssptr mzrss,char *tagname,char *smolname);`

> Local. Adds one pair of items, which are `tagname` and `smolname`, to the end of the moleculizer superstructure name hash. This expands the hash if needed. Returns the index of the newly added names for success or -1 if the hash had to be expanded and memory was insufficient.

`int mzraddtorxnhash(mzrssptr mzrss,char *mzrrxn,char *smolrxn);`

> Local. Adds one pair of items, which are `mzrrxn` and `smolrxn`, to the end of the moleculizer superstructure reaction name hash. This expands the hash if needed. Returns the index of the newly added names for success or -1 if the hash had to be expanded and memory was insufficient.

`void mzrNextSmolrxnName(mzrssptr mzrss,char *smolrxn);`

> Local. Returns a name, in `smolrxn`, for a new Smoldyn reaction. The name is simply "mzr#", where '#' is an integer that is incremented each time this function is called.

`int mzrMakeNameHash(simptr sim);`

> Local. This function updates the mapping between Smoldyn species and moleculizer's explicit names, which is stored in the moleculizer superstructure elements `tagname` and `smolname`. This function can work at initial set up or at other times. If called after initial set up, this replaces the entire hash with a new one. Returns 0 for success, 1 for inability to allocate memory, 2 for other bugs that shouldn't arise, 3 if moleculizer isn't sufficiently set up, 4 if Libmoleculizer is not defined in this build, or -1-i2 if two species have the same tagged name (an error, because it means the tagged name → Smoldyn name mapping is ambiguous); here `i2` is the index in the name hash of the prior name.

`int mzrssreadrules(simptr sim,ParseFilePtr *pfpptr,char *erstr);`

> Global. This reads the rules section of the input file into `mzrss->rules`. The string is pre-processed to remove comments, substitute "defines", handle "read_files", etc. Reading terminates when "end_rules" is encountered. Returns 0 for success or 1 for failure. If it returns with a failure, it also returns an error string with the failure. This function does not do any checking of the text within the rules file. It also does not parse any of the text within the rules file. This function is called by loadsim, in smolsim.c. Rule parsing is done with mzrssload.

`int mzrssload(simptr sim,char *erstr);`

> Local. This creates the moleculizer object (`mzrss->mzr`) and loads the rules that are in the superstructure rules element into it. Rules are parsed at this point. Returns 0 for success or 1 an error. Error messages are copied into the string `erstr`, which should be allocated to size `STRCHAR`.

`void mzrSetValue(mzrssptr mzrss,char *item,int i1);`

> Global. Sets various elements of `mzrss`. Enter item as "maxNetworkSpecies" to set that element to i1. Use a negative number for unlimited expansion and zero or a positive number for limited expansion. Enter `item` as "refspecies" to set that element to `i1`. Enter `item` as "expandall" to set that element to i1.

`int mzrsetupmoleculizer(simptr sim,char *erstr);`

> Global. Does whatever moleculizer setup operations are required, depending on the moleculizer superstructure condition. Returns 0 for success or 1 for failure. On failure, an error description is written to `erstr`, which should be allocated to size `STRCHAR`.

Core simulation functions

`int mzrAddSpeciesArrayToSim(simptr sim,species **species_array,int number_species);`

> Local. Takes in `number_species` moleculizer species, which are listed in `species_array`, and adds them both to the moleculizer superstructure name hash and to the Smoldyn list of species. Note that the type `species` is a `struct species_type` and that this struct is declared in libmzr_c_interface.h. Returns 0 for success or 1 for various failures, none of which should occur.

`int mzrCompareReactions(void *rxn1void,void *rxn2void);`

> Local. Compares reactions `rxn1void` and `rxn2void` and returns -1 if rxn1¡rxn2, 1 if rxn1¿rxn2, or 0 if they are equivalent. The lesser reaction is the one with more reactants, fewer mols in its reactants, and lower mass reactants. This sorting is essential for getting states correct.

`int mzrAddReactionArrayToSim(simptr sim,reaction **reactionlist,int number_reactions);`

 Local. Takes in `number_reactions` moleculizer reactions, which are listed in `reactionlist`, and adds them to the Smoldyn list of reactions. It does this through the `mzr_binaryRxnCreator` and `mzr_unaryRxnCreator` functions. Note that the type `reaction` is a `struct reaction_type` and that this struct is declared in libmzr_c_interface.h. Returns 0 for success or 1 for various failures, none of which should occur.

`int mzrExpandSpecies(simptr sim,int ident);`

 Global. This function expands the species with the Smoldyn species identity `ident`, using the Libmoleculizer function `expandSpeciesByTag`. If expansion happened, this downgrades the moleculizer superstructure condition to `SCparams` to address cascading effects. Returns 0 for success, 1 if Libmoleculizer cannot find the tagged name for the listed identity (which should not happen), or 2 if Libmoleculizer is not defined.

`int mzrExpandNetwork(simptr sim);`

 Local. Expands the entire rule-based network using Libmoleculizer's function `expandNetwork`. This function ignores the `maxNetworkSpecies` element of the moleculizer superstructure. It may take a very long time to run and may not return at all if the network is infinite. If expansion happened, this downgrades the moleculizer superstructure condition to `SCparams` to address cascading effects. Returns 0 for success, 1 if Libmoleculizer returns an error code, or 2 if Libmoleculizer is not defined.

`int mzrExpandUnexpandedSpecies(simptr sim);`

 Local. This function expands all unexpanded species that are listed in Smoldyn's molecule superstructure. It does this with the `mzrExpandSpecies` function. It returns 0 on success or 1 for various errors, none of which should occur.

`int mzrAddRxn(simptr sim,char *name,int order,int *reactants,int *products,int`
`  nprod,double rate);`

 Local. Adds a reaction named `name` to the Smoldyn simulation. This reaction has `order` reactant identities listed in `reactants`, `nprod` product identities listed in `products`, and a rate constant of `rate`. Returns 0 for success or 1 if Smoldyn's `RxnAddReaction` couldn't find memory for the reaction.

 This function deals with reactant and product states. The rule is that if no default states have been set at all, then all states, for both reactants and products, are only set to solution state. On the other hand, if a default state exists for that species, then that state is used....

5.11 Complexes (not written yet)

It's becoming increasingly apparent that Smoldyn needs to support macromolecular complexes, and also that Libmoleculizer isn't going to be adequate. In part, Libmoleculizer doesn't have all of the necessary features and in part, Libmoleculizer is a horribly complicated mess, which doesn't even work at present. 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]

5.12   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]
   } *graphicsssptr;
```

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.

5.13   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
  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
  graphicsssptr 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.

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.

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	w suppresses
		structure condition not updated	
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.

`void 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.

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 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.

`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`, or 10 for simulation stopped because the time equals or exceeds the break time. 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.

5.14 Commands (functions in smolcmd.c)

Externally accessible function

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`. See below for how to add a new command.

`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

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.

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.

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 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.

`enum CMDcode cmdspeciesstreamcountheader(simptr sim,cmdptr cmd,char *line2);`
    Prints out all libmoleculizer species stream names to a single line of text. This function requires
    memory allocation and freeing on each function call, which libmoleculizer does.

`enum CMDcode cmdspeciesstreamcount(simptr sim,cmdptr cmd,char *line2);`
    Prints the time, followed the number of molecules in each species stream, all on a single line. Note that
    molecules may be in no species streams, one species stream, or multiple species streams. This function
    requires memory allocation and freeing on each function call, which libmoleculizer does.

`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 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 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.

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 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 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 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 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.

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.

5.15   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 6

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.

6.1 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
```

6.2 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.

> <u>molsupdate</u>
>   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 indicies
sorts and compacts live lists (calls boxremovemol and boxaddmol)
moves molecules from resurrected to reborn lists (calls boxaddmol)
resets sortl indicies

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

6.3 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).

Chapter 7

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.

Chapter 8

The wish/ to do list

8.1 Bugs and issues to fix

· Libmoleculizer needs to be checked for reaction rate parameters for reversible reactions, especially when used in on-the-fly mode. My concern is that Smoldyn needs to know about reversibility when it computes parameters, and so it might get things wrong if things initially appear to be irreversible.

· 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.

· Finish pthreads additions, or just kill pthreads support.

· If `time_now` is not start time, then all commands from start time to current time get run. They shouldn't.

· The `set` command needs more work to be functional and bug-free. It might be finished as of version 2.23, but needs to be tested.

· 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.

8.2   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.

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.

· 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.

· Macromolecular complexes. Lots of people want these features. At a minimum, we really need to get Libmoleculizer to work. Eventually, it might be worthwhile to think of more sophisticated rule-based modeling methods. In particular, it would be good to be able to capture the spatial arrangement, dynamics, and allostery of the chemotaxis receptor cluster. Also, the allostery and dynamics of the CamKII protein (a dodecamer comprised of two coaxial circular hexamers, plus lots of phosphorylation sites that allosterically interact with each other).

· 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.