Libsmoldyn User's Manual

for Smoldyn version 2.62

Steve Andrews

Contents

Chapter 1

About Libsmoldyn

Libsmoldyn is a C, C++, and Python interface to the Smoldyn simulator. 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 (API) 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 was originally written as a C API, but one that should work with C++ as well. Then, in 2019 and 2020, Dilawar Singh added Python bindings to it, which are still quite new but generally work well.

Chapter 2

Use with Python

2.1 Installing

It's best to install Smoldyn twice. First, download the latest package from `http://www.smoldyn.org` and run the install script. This will install the stand-alone program, the C/C++ libraries, and the BioNetGen code, and will also give you the documentation and example files.

Next, install the Python components from PyPI with:

```
pip install smoldyn
```

If this doesn't work, then a different possibility is:

```
python3 -m pip install smoldyn --user --pre
```

Either of these should install the smoldyn nightly package, from: `https://pypi.org/project/smoldyn/`. For Windows, Python is often called "py", so try this if the above doesn't work (or your computer asks you to download Python from the Windows store, which probably isn't necessary). For those who are curious, the "-m" option means to run pip as a python module, the "–user" option means to install to the user directory, and the "–pre" option means to include pre-release versions. Feel free to try more or fewer options, of course. Also, some other useful pip options are: "pip uninstall smoldyn", "pip cache purge" (pip installs from its cached version if it already has one, so this is useful if you want to force it to download the online version), and "pip search smoldyn".

Alternatively, the Mac distribution, downloaded from `http://www.smoldyn.org/downloads/` comes with a Python wheel, called something like smoldyn-2.62-py3-none-any.whl. After installing with the software with the "install.sh" script, you should be able to write "pip install smoldyn...whl" and that will install it for you as well.

After installing, it might just run. If not, then the next challenge is to get your system to know that it's allowed to run the files. There are a few ways to do this. (1) You can navigate to one directory above where the "__init__.py" file gets stored and work from there. To try this out, go there and start Python; then try `import smoldyn` and see if it works. (2) You can set `PYTHONPATH` environment variable to the install location. However, this only works for the current command-line session and needs to be reset for the next one. To see its current value, for Mac or Linux, enter `echo $PYTHONPATH`. (3) You can modify your sys.path variable to include a path to the install location, which then lasts for new command-line sessions as well. To see its current value, start Python and enter: `import sys; print(sys.path)`.

2.2 Current limitations

The Python interface has a few limitations that users should be aware of. We are working on fixing them.

Rule-based modeling. Rule-based modeling support, including both BioNetGen and Smoldyn wildcards, has not been added to Libsmoldyn yet, including its Python API.

Hybrid simulation. Smoldyn has built-in support for multiscale simulation in which part of the model volume is simulated with particle-based methods and part with spatial Gillespie methods. These parts are

typically adjacent to each other but can also overlap. This functionality has not been added to Libsmoldyn yet, including its Python API.

Python quits after simulation if graphics are used. A particular benefit of running Smoldyn through the Python interface is that the simulation and data analysis code can be included in the same Python file. Unfortunately, Smoldyn displays its graphical output using the OpenGL glut library, which never returns control to the calling program (Smoldyn) once it has been started. As a result, quitting OpenGL leads to a termination of the Python session, so no further Python code gets run. The only good solution for now is to display graphics while developing a simulation, and then re-run simulations with graphics turned off when collecting quantitative data. Note that the random number seed can be set to the same value in each case so that the simulation will be exactly the same.

TIFF output doesn't work on some computers. For some reason, Smoldyn cannot output TIFF files when run through the Python interface, at least on some computers. The Smoldyn code itself runs just fine, but it calls the `TIFFOpen` function in the libtiff library, which somehow refuses to open a new TIFF file. Alternative options for exporting graphics are: (1) rewrite the model without the Python interface, (2) save the current model and state using the savesim command, and then run that using the stand-alone Smoldyn software (i.e. the Smoldyn application, without the Python interface), (3) pause the simulation and capture the graphics using a screen grab, (4) use the VTK graphical output option. Yet another option is to convince a Smoldyn developer to add png export support for Smoldyn, such as with the "Tiny PNG Output" project, which looks fairly easy but hasn't been done yet.

2.3 Python API example

The following example, `template.py`, is from the examples directory in the Smoldyn download. It simulates a Michaelis-Menten reaction in which substrates and products diffuse freely within a circular membrane (it's simulated in two dimensions, for simplicity) and enzymes are bound to the membrane.

```
1  """Template for writing Smoldyn model in Python
2
3  Use standard docstring to list basic file information here, including your
4  name, the development date, what this file does, the model name if you want
5  one, units used, the file version, distribution terms, etc.
6
7  Enzymatic reactions on a surface, by Steve Andrews, October 2009. Modified by
8  Dilawar Singh, 2020. This model is in the public domain. Units are microns and seconds.
9  The model was published in Andrews (2012) Methods for Molecular Biology, 804:519.
10 It executes a Michaelis-Menten reaction within and on the surface of a 2D circle.
11 """
12
13 __author__ = "Dilawar␣Singh"
14 __email__ = "dilawars@ncbs.res.in"
15
16 import smoldyn
17
18 # Model parameters
19 K_FWD = 0.001     # substrate-enzyme association reaction rate
20 K_BACK = 1        # complex dissociation reaction rate
21 K_PROD = 1        # complex reaction rate to product
22
23 # Simulation starts with declaring a Simulation object with the system boundaries.
24 s = smoldyn.Simulation(low=[-1, -1], high=[1, 1])
25
26 # Molecular species and their properties
27 # Species: S=substrate, E=enzyme, ES=complex, P=product
28 # Type `help(smoldyn.Species)` in Python console to see all parameters.
29 S = s.addSpecies("S", difc=3, color=dict(all="green"), display_size=dict(all=0.02))
30 E = s.addSpecies("E", color=dict(all="darkred"), display_size=dict(all=0.03))
31 P = s.addSpecies("P", difc=3, color=dict(all="darkblue"), display_size=dict(all=0.02))
32 ES = s.addSpecies("ES", color=dict(all="orange"), display_size=dict(all=0.03))
33
34 # Surfaces in and their properties. Each surface requires at least one panel.
35 # Add action to `both` faces for surface. You can also use `front` or `back`
```

```
36 # as well. Here, 'all' molecules reflect at both surface faces.
37 sph1 = smoldyn.Sphere(center=(0, 0), radius=1, slices=50)
38 membrane = s.addSurface("membrane", panels=[sph1])
39 membrane.setAction('both', [S, E, P, ES], "reflect")
40 membrane.setStyle('both', color="black", thickness=1)
41
42 # Define a compartment, which is region inside the 'membrane' surface.
43 inside = s.addCompartment(name="inside", surface=membrane, point=[0, 0])
44
45 # Chemical reactions. Here, E + S <-> ES -> P
46 r1 = s.addBidirectionalReaction(
47     "r1", subs=[(E,"front"), (S,"bsoln")], prds=[(ES,"front")], kf=K_FWD, kb=K_BACK)
48 r1.reverse.productPlacement("pgemmax", 0.2)
49
50 r2 = s.addReaction(
51     "r2", subs=[(ES, "front")], prds=[(E, "front"), (P, "bsoln")], rate=K_PROD)
52
53 # Place molecules for initial condition
54 inside.addMolecules(S, 500)
55 membrane.addMolecules((E, "front"), 100)
56
57 # Output and other run-time commands
58 s.setOutputFile('templateout.txt', True)
59 s.addCommand(cmd="molcountheader␣templateout.txt", cmd_type="B")
60 s.addCommand(cmd="molcount␣templateout.txt", cmd_type="N", step=10)
61
62 s.setGraphics(
63     "opengl_good", bg_color="white", frame_thickness=1,
64     text_display=["time", S, (E, "front"), (ES, "front"), P] )
65 s = s.run(stop=10, dt=0.01)
```

1-11. The file starts with a docstring, which is a useful way to provide information about the model, authors, etc. Smoldyn does not handle units at all, so it's the user's responsibility to make sure that all units are consistent with each other. The best approach is simply to make sure that all lengths use the same units, such as nanometers or microns, and all times use the same units, such as milliseconds. This also applies to derived units, such as volumes and rate constants.

13-14. Entering the author and email address with double underscores follows good Python form.

16. Import the Smoldyn Python API with `import smoldyn`.

18-21. This file defines some variables, which are just regular variables and not part of Smoldyn at all. The file uses upper case variable names to stay consistent with the stand-alone Smoldyn example file with the same name, but this isn't really the best convention in Python.

23-24. All models start by creating a Simulation object, which includes the boundaries of the system space; the format is . The dimensionality of the space, whether 1D, 2D, or 3D, is inferred from the number of boundary coordinates given. Smoldyn actually tracks molecules outside of this volume, too, but it runs most efficiently if most of the action in the simulation occurs within the system volume. It's possible to create multiple simulation objects if you want.

26-32. Add species to a simulation with *sim*.`addSpecies`. Each species is required to have a name. The diffusion coefficient, given with `difc`, defaults to a value of 0, such as for the "E" species. The color, which is only used for graphical output, can often be given with a simple assignment, such as `color='green'`. However, that's not used here because these molecules can have multiple states, including in solution or bound to the surface. Instead, the colors are specified here with `dict()` functions, showing that these colors apply to all of the molecule states. Similarly, the `display_size` argument is only used for graphical output, and is given with `dict()` functions here because of the multiple states.

34-40. This model includes surface, called "membrane". This surface is composed of one circular panel, which is called a sphere here because Smoldyn names its panels based on analogous 3D shapes. The panel is created first by defining its center and radius. The "slices" parameter refers to how many sides should be drawn on the circle for graphical output, since Smoldyn doesn't actually draw perfect circles. Internally, a circle or sphere is defined using the mathematical definition of a circle or sphere, so it doesn't have flat sides. Each surface has a front and back face; for a spherical panel, as used here, the front face is the outside by default. After defining the panel, the code defines a surface and the panel is added to it. Then, the surface actions are defined, meaning what happens to molecules that diffuse into it. In this case, for both the front

and back faces of the surface, any molecule of the given list of species (which happens to be all of them) is reflected back toward where it came from. Finally, the surface drawing style is defined, here showing that both the front and back faces should be black and have a thickness of 1 pixel.

42-43. Compartments are regions of space that are bounded by surfaces, in this case "membrane". Also, they have "interior-defining points", which define which side of the surface represents the inside of the compartment.

45-51. Chemical reactions have names, substrates, products, and rate constants. Here, each of the substrates and products are listed with both species and states, although states are assumed to be solution if not given. Because the front face of a spherical panel is the outside, the fact that the substrates enter from the back side of the surface is specified by specifying that they have "bsoln" states, meaning in solution on the back side. The products are similar. For reversible reactions, there is always some probability that the two product molecules of one of the reaction directions will diffuse back together and bind back together again, which is called a geminate recombination. It's not essential, but it's good practice to tell Smoldyn how likely this recombination should be. Here, it's set to a maximum probability of 0.2, which generally works well.

53-55. Add molecules to the simulation with the `addMolecules` function, where its used here both for adding molecules into the compartment and onto the surface.

57-60. Run-time commands are used for manipulating or observing the simulated system as it runs, effectively acting as the experimenter. In this case, an output file is declared, called "templateout.txt". Then, right before the simulation runs, given by command type 'B', the command called "molcountheader" is run with parameter "templateout.txt", which is the file name that it should write to. This prints out a header for the molcount command, which is just a list of species names. The other command is type 'N', which means that it runs every n time steps, which is this case is every 10 steps. It is the "molcount" command with parameter "templateout.txt", which prints out the number of each molecule type to the same file.

62-64. Smoldyn run-time graphics show the simulation as it runs, always using the OpenGL library. Here, the "opengl_good" method is used, which produces nice output but without any lighting effects. The background color is white and the frame thickness is 1. Also, this statement says to show text on the graphics window listing the time and numbers of the different molecule types.

65. Finally, the simulation is run, here for 10 time units in steps of 0.01 time units.

2.4 Callback functions

Python offers a callback function that enables Smoldyn setup functions to be called repeatedly and automatically.

For example, suppose we have the following function, `computeVm`, which generates a noisy value using the current time, `t`, and a list of arguments, `args`. Also, we have a molecular species, `ca`. The output of the `computeVm` function can be connected to the ca.difc parameter, which is called every 10th step in this case.

```
def computeVm(t, args):
    x, y = args
    return math.sin(t) + x * y + random.random()
...
import smoldyn
sim = smoldyn.simulation(low=[0, 0], high=[10, 10])
ca = sim.addSpecies('ca', difc=1, color='blue', display_size=1)
...
sim.connect(func = computeVm, target = 'ca.difc', step=10, args=[1,2.1])
```

Both source and target in the connect function must be global variables. In the example below, a global variable is made before it is used it in connect because otherwise there would be a runtime error.

```
import smoldyn
import random

a = None

def new_dif(t, args):
```

```
    global a
    x, y = args
    print(a.difc)
    return t + random.random()

def test_connect():
    global a
    sim = smoldyn.Simulation(low=(0, 0), high=(10, 10))
    a = sim.addSpecies('a', color='red', difc=1)
    sim.connect(func = new_dif, target = 'a.difc', step=10, args=[0, 1])
    sim.run(100, 1)
    print('All␣done')

test_connect()
```

The following example shows that connect accepts a function.

```
def new_dif(t, args):
    global a, avals
    x, y = args
    # note that b.difc is not still updated.
    avals.append((t, a.difc["soln"]))
    return x * math.sin(t) + y

def update_difc(val):
    global a
    a.difc = val

def test_connect():
    global a, avals
    sim = smoldyn.Simulation(low=(0, 0), high=(10, 10))
    a = sim.addSpecies('a', color='black', difc=0.1)
    sim.connect(new_dif, update_difc, step=10, args=[1, 1])
    sim.run(100,1)
    for a, b in zip(avals[1:], expected_a[1:]):
        print(a, b)
        assert math.isclose(a[1], b[1], rel_tol=1e-6, abs_tol=1e-6), (a[1], b[1])

test_connect()
```

Additional examples of `connect` are included in examples/S99_more/change_env.py, which simulates a pre-synaptic bouton with N synaptic vesicles. These vesicles fuse with the bottom of the bouton (red surface). Upon fusion, one vesicle releases 1000 neurotransmitters which decay with time-constant τ. The rate of release is controlled by a function that is set by `connect`. The function generates a spike 0 or 1; if the value is 1, the rate is set to 1000, else it is 0.

Chapter 3

Use with C/C++

3.1 Compiling

Header files

To enable a C or C++ program to call Libsmoldyn, it has to include the Libsmoldyn header file. Libsmoldyn comes with one header file, libsmoldyn.h, which has function declarations for all of the Libsmoldyn functions. For most Libsmoldyn applications, this is the only header file that you will need to include. For Mac and Linux, it is typically installed to /usr/local/include. This is one of the standard system paths, so include it with

```
#include <libsmoldyn.h>
```

If the libsmoldyn.h header file is in some other directory or if your system isn't seeing its path as a system path, then include the file using double quotes rather than angle brackets and/or include more information about the path. For example, #include "/user/local/include/libsmoldyn.h".

Libsmoldyn.h calls a second header file, smoldyn.h, which is also typically installed to /usr/local/include/. If you plan to access the Smoldyn data structure directly, then you will also need to include it with #include <smoldyn.h>. In general, it is safe to read from this data structure but it can be dangerous to write to it unless you really know what you are doing. Also, working with this data structure directly bypasses one of the benefits of using Libsmoldyn, which is that the interface should be relatively immune to future Smoldyn developments, because different aspects of the internal data structure get changed once in a while.

The smoldyn.h header calls yet another header file, smoldynconfigure.h, which is also installed by default in /usr/local/include/. That file is automatically generated by the build system. It describes what Smoldyn components are included in the build, what system the build was compiled for, etc. This might be helpful to include for some applications.

Compiling 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.

3.2 Linking

Linking the different object files together to create an executable that actually runs is often one of the greatest frustrations of using software libraries. It should be easy but usually isn't.

The Libsmoldyn library 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. Dynamic linking is somewhat more elegant in that it doesn't create unnecessary copies of the compiled code. It can also be easier. On the other hand, it's less convenient if you plan to distribute your software, because then you need to make sure that you distribute the Libsmoldyn header file and library code along with your own software. Also, I can only get the gdb debugger to help find errors within Libsmoldyn if the library is statically linked.

The Libsmoldyn static library is called libsmoldyn_static.a and the Libsmoldyn dynamic library is called libsmoldyn_shared.so (on Linux; the .so suffix is replaced by .dylib on a Mac and by .dll on Windows). By default, these libraries are installed to /usr/local/lib/.

Linking examples

Following are several example for static and dynamic linking. They are shown for C; if you use C++, then link with g++ instead of gcc. The linking options for Smoldyn compiled with OpenGL are shown for Macintosh; these lines are simpler for other systems.

I have had a hard time getting static linking working on a Mac, although apparently it works fine on Ubuntu. The problem is that it doesn't find the standard C++ library. The solution is to build the Smoldyn library without NSV, so that the standard C++ library isn't needed. I also commented out a few "throw" statements from smolsim.c and libsmoldyn.c for this purpose.

Static link, no OpenGL:

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

Static link, with OpenGL:

```
gcc testcode.o /usr/local/lib/libsmoldyn_static.a -I/System/Library/Frameworks/OpenGL.framework/Headers -I/
```

Dynamic link, no OpenGL:

```
gcc testcode.o -o testcode -lsmoldyn_shared
```

Dynamic link, with OpenGL:

```
gcc test1.o -L/usr/local/lib -I/System/Library/Frameworks/OpenGL.framework/Headers -I/System/Library/Fram
```

3.3 Memory management

None of the Libsmoldyn 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. That memory clearly needs to be freed, by the library user, when it is done being used. When the simulation is complete, the simulation data structure should be freed, which will automatically free all substructures in the process.

Nearly all strings are fixed at `STRCHAR` characters, where this constant is defined in string2.h to 256 characters.

3.4 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 that calls Libsmoldyn, 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 {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

3.5 Error checking internal to libsmoldyn.c

This section 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 either the `LCHECK` or `LCHECKNT` macros. In both cases, the macro format is `LCHECK(`*condition*`, funcname, `*error_code*`, "`*message*`");`. The macros check that the test *condition* is true, and calls either `smolSetError` or `smolSetErrorNT` to deal with it if not. The *message* should be a descriptive message that is under 256 characters in length. Use the regular version (not the "no throw" or "NT") version for errors that arise within the function, and the "NT" version for errors that arise is subroutines of the function, so that only a single error message is displayed to the output.

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.

Chapter 4

C/C++ and low-level Python API

4.1 Quick function guide

Most 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". The Libsmoldyn functions are available either through the C/C++ API or through the low-level Python API, with essentially identical input styles.

Statement	Function
About the input	
#	N/A
/* ... */	N/A
read_file	`LoadSimFromFile`
	`ReadConfigString`
end_file	N/A
quit_at_end	
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
variable	N/A
N/A	`SetError`
N/A	`GetError`
N/A	`ClearError`
N/A	`SetDebugMode`
N/A	`ErrorCodeToString`
Space and time	
dim	`NewSim`
boundaries	`NewSim`
	`SetBoundaryType`
low_wall	`NewSim`
	`SetBoundaryType`
high_wall	`NewSim`
	`SetBoundaryType`

time_start	`SetSimTimes`
	`SetTimeStart`
time_stop	`SetSimTimes`
	`SetTimeStop`
time_step	`SetSimTimes`
	`SetTimeStep`
time_now	`SetTimeNow`
Molecules	
species	`AddSpecies`
species_group	
N/A	`GetSpeciesIndex`
N/A	`GetSpeciesName`
difc	`SetSpeciesMobility`
difc_rule	
difm	`SetSpeciesMobility`
difm_rule	
drift	`SetSpeciesMobility`
drift_rule	
surface_drift	
surface_drift_rule	
mol	`AddSolutionMolecules`
surface_mol	`AddSurfaceMolecules`
compartment_mol	`AddCompartmentMolecules`
molecule_lists	`AddMolList`
mol_list	`AddSpecies`
	`SetMolList`
mol_list_rule	
N/A	`GetMolListIndex`
N/A	`GetMolListName`
max_mol	`SetMaxMolecules`
N/A	`GetMoleculeCount`
Graphics	
graphics	`SetGraphicsParams`
graphic_iter	`SetGraphicsParams`
graphic_delay	`SetGraphicsParams`
quit_at_end	
frame_thickness	`SetFrameStyle`
frame_color	`SetFrameStyle`
grid_thickness	`SetGridStyle`
grid_color	`SetGridStyle`
background_color	`SetBackgroundStyle`
display_size	`SetMoleculeStyle`
color	`SetMoleculeStyle`
tiff_iter	`SetTiffParams`
tiff_name	`SetTiffParams`
tiff_min	`SetTiffParams`
tiff_max	`SetTiffParams`
light	`SetLightParams`
text_color	`SetTextStyle`
text_display	`AddTextDisplay`
Run-time commands	
output_root	`SetOutputPath`
output_files	`AddOutputFile`
output_data	

output_precision	
append_files	`AddOutputFile`
output_file_number	`AddOutputFile`
output_format	
cmd	`AddCommand`
	`AddCommandFromString`
Surfaces	
start_surface	`AddSurface`
new_surface	`AddSurface`
* name	`AddSurface`
N/A	`GetSurfaceIndex`
N/A	`GetSurfaceName`
* action	`SetSurfaceAction`
* action_rule	
* rate	`SetSurfaceRate`
* rate_rule	
* rate_internal	`SetSurfaceRate`
* rate_internal_rule	
* neighbor_action	
* color	`SetSurfaceFaceStyle`
	`SetSurfaceEdgeStyle`
* thickness	`SetSurfaceEdgeStyle`
* stipple	`SetSurfaceEdgeStyle`
* polygon	`SetSurfaceFaceStyle`
* shininess	`SetSurfaceFaceStyle`
* panel	`AddPanel`
N/A	`GetPanelIndex`
N/A	`GetPanelName`
* jump	`SetPanelJump`
* neighbors	`AddPanelNeighbor`
* unbounded_emitter	`AddSurfaceUnboundedEmitter`
* end_surface	N/A
epsilon	`SetSurfaceSimParams`
margin	`SetSurfaceSimParams`
neighbor_dist	`SetSurfaceSimParams`
Compartments	
start_compartment	`AddCompartment`
new_compartment	`AddCompartment`
* name	`AddCompartment`
N/A	`GetCompartmentIndex`
N/A	`GetCompartmentName`
* surface	`AddCompartmentSurface`
* point	`AddCompartmentPoint`
* compartment	`AddCompartmentLogic`
* end_compartment	N/A
Reactions	
reaction	`AddReaction`
N/A	`GetReactionIndex`
N/A	`GetReactionName`
reaction compartment=...	`SetReactionRegion`
reaction surface=...	`SetReactionRegion`
reaction_rule	
reaction_rate	`AddReaction`
	`SetReactionRate`

reaction_multiplicity	
confspread_radius	`SetReactionRate`
binding_radius	`SetReactionRate`
reaction_probability	`SetReactionRate`
reaction_chi	
reaction_production	`SetReactionRate`
product_placement	`SetReactionProducts`
expand_rules	
reaction_serialnum	
reaction_intersurface	
reaction_log	
reaction_log_off	
Ports	
start_port	`AddPort`
new_port	`AddPort`
* name	`AddPort`
N/A	`GetPortIndex`
N/A	`GetPortName`
* surface	`AddPort`
* face	`AddPort`
* end_port	N/A
N/A	`AddPortMolecules`
N/A	`GetPortMolecules`
Rule-based modeling with BioNetGen	
start_bng	
end_bng	
name	
BNG2_path	
multiply unimolecular_rate	
multiply bimolcular_rate	
monomer	
monomers	
monomer_difc	
monomer_display_size	
monomer_color	
monomer_state	
expand_rules	
Lattices	
start_lattice	
* name	
* type	
* port	
* boundaries	
* lengthscale	
* species	
* make_particle	
* reaction	
* reaction move	
* mol	
* end_lattice	
Simulation settings	
random_seed	`SetRandomSeed`
accuracy	not supported
molperbox	`SetPartitions`

boxsize	`SetPartitions`
gauss_table_size	not supported
epsilon	`SetSurfaceSimParams`
margin	`SetSurfaceSimParams`
neighbor_dist	`SetSurfaceSimParams`
Libsmoldyn actions	
N/A	`UpdateSim`
N/A	`RunTimeStep`
N/A	`RunSim`
N/A	`RunSimUntil`
N/A	`FreeSim`
N/A	`DisplaySim`
N/A	`PrepareSimFromFile`

4.2 Enumerations

In C, enumerations are already defined, so they can be used as is. Here is an example of using an enumerated error code as an argument,

```
smolErrorCodeToString(ECwarning, mystring)
```

In Python, enumerations are most easily dealt with by defining a variable for the enumerated list and then choosing from it. Here is an example,

```
import smoldyn._smoldyn as S
EC=S.ErrorCode
S.errorCodeToString(EC.warning, mystring)
```

Surface actions (SrfAction)

Statement	Libsmoldyn	Python	Notes
reflect	SAreflect	reflect	
transmit	SAtrans	trans	
absorb	SAabsorb	absorb	
jump	SAjump	jump	
port	SAport	port	
multiple	SAmult	mult	multiple actions
N/A	SAno	no	static surface-bound molecules
N/A	SAnone	none	none of the other options
N/A	SAadsorb	adsorb	internal use only
N/A	SArevdes	revdes	internal use only
N/A	SAirrevdes	irrevdes	internal use only
N/A	SAflip	flip	internal use only

Molecule state (MolecState)

Statement	Libsmoldyn	Python	Notes
soln	MSsoln	soln	
front	MSfront	front	
back	MSback	back	
up	MSup	up	
down	MSdown	down	
bsoln	MSbsoln	bsoln	pseudo-state for surface interactions
all	MSall	all	for model creation by user
N/A	MSnone	none	internal use only

Panel face (PanelFace)

Statement	Libsmoldyn	Python	Notes
front	PFfront	front	
back	PFback	back	
N/A	PFnone	none	internal use only
both	PFboth	both	for model creation by user

Panel shape (PanelShape)

Statement	Libsmoldyn	Python	Notes
rect	PSrect	rect	rectangle
tri	PStri	tri	triangle
sph	PSsph	sph	sphere
cyl	PScyl	cyl	cylinder
hemi	PShemi	hemi	hemisphere
disk	PSdisk	disk	disk
all	PSall	all	for model creation by user
N/A	PSnone	none	internal use only

Libsmoldyn error code (ErrorCode)

Statement	Libsmoldyn	Python	Notes
N/A	ECok	ok	value 0
N/A	ECnotify	notify	value -1
N/A	ECwarning	warning	value -2
N/A	ECnonexist	nonexist	value -3
N/A	ECall	all	value -4
N/A	ECmissing	missing	value -5
N/A	ECbounds	bounds	value -6
N/A	ECsyntax	syntax	value -7
N/A	ECerror	error	value -8
N/A	ECmemory	memory	value -9
N/A	ECbug	bug	value -10
N/A	ECsame	same	value -11
N/A	ECwildcard	wildcard	value -12

4.3 Miscellaneous

GetVersion

C/C++: `double smolGetVersion(void)`
Python: `float getVersion()`
Returns the Smoldyn version number.

4.4 Errors

SetError

C/C++: `void smolSetError(const char *errorfunction, enum ErrorCode errorcode, const char *errorstring)`
C/C++: `void smolSetErrorNT(const char *errorfunction, enum ErrorCode errorcode, const char *errorstring)`
Python: N/A
These functions are 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`. For the regular version without the "NT", if the library is being run in debug mode, then this function prints the notification, warning, or error out to stderr. It would also ideally throw exceptions if the error code is more severe than the `LibThrowThreshold` value, but this throwing doesn't work at present because throwing exceptions to the host code is incompatible with static linking.

The "NT" version is the "no throw" version, which is the same as the regular version but doesn't display messages to stderr and doesn't throw exceptions. In general, library functions should call `smolSetError` for errors caught by that function itself, and `smolSetErrorNT` for errors caught by subroutines of that function, so that each error only leads to a single call of `smolSetError`.

GetError

C/C++: `enum ErrorCode smolGetError(char *errorfunction, char *errorstring, int clearerror)`
Python: N/A
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.

ClearError

C/C++: `void smolClearError(void)`
Python: N/A
Clears any error condition.

SetDebugMode

C/C++: `void smolSetDebugMode(int debugmode)`
Python: `setDebugMode(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`.

ErrorCodeToString

C/C++: `char* smolErrorCodeToString(enum ErrorCode erc, char *string)`
Python: `str errorCodeToString(enum ErrorCode erc, str 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". To do: The string is not needed or used in the Python version.

4.5 Sim structure

NewSim

Python: *sim* = `Simulation(low = List[float],high = List[float])`
Python: `simptr newSim(int dim, List[float] lowbounds, List[float] highbounds)`
C/C++: `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.

UpdateSim

Python: N/A
Python: `ErrorCode updateSim()`
C/C++: `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.
Python low-level to do: doesn't work. Python wants no argument, but Libsmoldyn then complains about no argument.

RunTimeStep
C/C++: `enum ErrorCode smolRunTimeStep(simptr sim)`
Python: `ErrorCode runTimeStep()`
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.
Python to do: doesn't work. Python wants no argument, but Libsmoldyn then complains about no argument.

RunSim
C/C++: `enum ErrorCode smolRunSim(simptr sim)`
Python: `ErrorCode runSim()`
Python: `run(stop=None, start=None, step=None)`
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.
Python to do: doesn't work. Python wants no argument, but Libsmoldyn then complains about no argument.

RunSimUntil
C/C++: `enum ErrorCode smolRunSimUntil(simptr sim, double breaktime)`
Python: `ErrorCode runSimUntil(float breaktime)`
Python: `runUntil(t, dt)`
Runs the simulation either until it terminates or until the simulation time equals or exceeds `breaktime`.
Python to do: doesn't work. Python wants no argument, but Libsmoldyn then complains about no argument.

FreeSim
C/C++: `enum ErrorCode smolFreeSim(simptr sim)`
Python: `ErrorCode freeSim()`
Frees the simulation data structure.

DisplaySim
C/C++: `enum ErrorCode smolDisplaySim(simptr sim)`
Python: `ErrorCode displaySim()`
Displays all relevant information about the simulation system to stdout.

4.6 Read configuration file

PrepareSimFromFile
C/C++: `simptr smolPrepareSimFromFile(char *filepath, char *filename, char *flags)`
Python: `simptr prepareSimFromFile(str filename, str 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`.

LoadSimFromFile
C/C++: `enum ErrorCode smolLoadSimFromFile(char *filepath, char *filename, simptr *simpointer, char *flags)`
Python: `ErrorCode loadSimFromFile(str filename, str 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.

ReadConfigString

C/C++: `enum ErrorCode smolReadConfigString(simptr sim, char *statement, char *parameters)`

Python: `ErrorCode readConfigString(str statement, str 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.

4.7 Simulation settings

SetSimTimes

C/C++: `enum ErrorCode smolSetSimTimes(simptr sim, double timestart, double timestop, double timestep)`

Python: `ErrorCode setSimTimes(float timestart, float timestop, float timestep)`

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

SetTimeStart

C/C++: `enum ErrorCode smolSetTimeStart(simptr sim, double timestart)`

Python: `ErrorCode setTimeStart(float timestart)`

Sets the simulation starting time.

SetTimeStop

C/C++: `enum ErrorCode smolSetTimeStop(simptr sim, double timestop)`

Python: `ErrorCode setTimeStop(float timestop)`

Sets the simulation stopping time.

SetTimeNow

C/C++: `enum ErrorCode smolSetTimeNow(simptr sim, double timenow)`

Python: `ErrorCode setTimeNow(float timenow)`

Sets the simulation current time.

SetTimeStep

C/C++: `enum ErrorCode smolSetTimeStep(simptr sim, double timestep)`

Python: `ErrorCode setTimeStep(float timestep)`

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

SetRandomSeed

C/C++: `enum ErrorCode smolSetRandomSeed(simptr sim, double seed)`

Python: `ErrorCode setRandomSeed(int 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.

SetAccuracy

C/C++: not supported

Python: `accuracy(accuracy: float)`

Sets or gets the simulation accuracy.

SetPartitions

C/C++: `enum ErrorCode smolSetPartitions(simptr sim, char *method, double value)`
Python: `ErrorCode setPartitions(str method, float value)`
Python: `Partition(name: str, value: float)`
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.

The second Python option is its own class. I think this should be removed because partitions aren't physical objects and so don't really make sense here.

MoleculePerBox

Python: `MoleculePerBox(size: float)`
This is only available in Python. Again, I think this should be removed because partitions aren't physical objects.

Box

Python: `Box(size: float)`
This is only available in Python. Again, I think this should be removed because partitions aren't physical objects.

4.8 Graphics

SetGraphicsParams

C/C++: `enum ErrorCode smolSetGraphicsParams(simptr sim, char *method, int timesteps, double delay)`
Python: `ErrorCode setGraphicsParams(str method, int timesteps, int delay)`
Python: `setGraphics(method: str, timestep: int, delay: int = 0)`
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.

SetTiffParams

C/C++: `enum ErrorCode smolSetTiffParams(simptr sim, int timesteps, char *tiffname, int lowcount, int highcount)`
Python: `ErrorCode setTiffParams(int timesteps, str 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.

SetLightParams

C/C++: `enum ErrorCode smolSetLightParams(simptr sim, int lightindex, double *ambient, double *diffuse, double *specular, double *position)`
Python: `ErrorCode smolSetLightParams(int lightindex, List[float] ambient, List[float]`

`diffuse, List[float] specular, List[float] 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`.

SetBackgroundStyle
C/C++: `enum ErrorCode smolSetBackgroundStyle(simptr sim, double *color)`
Python: `ErrorCode setBackgroundStyle(string color)`
Sets the color of the graphics display background. `color` is a 4-value vector with red, green, blue, and alpha values (or, in Python, a color word).

SetFrameStyle
C/C++: `enum ErrorCode smolSetFrameStyle(simptr sim, double thickness, double *color)`
Python: `ErrorCode setFrameStyle(float thickness, string 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 (or, in Python, a color word).

SetGridStyle
C/C++: `enum ErrorCode smolSetGridStyle(simptr sim, double thickness, double *color)`
Python: `ErrorCode setGridStyle(float thickness, string 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 (or, in Python, a color word).

SetTextStyle
C/C++: `enum ErrorCode smolSetTextStyle(simptr sim, double *color)`
Python: `ErrorCode setTextStyle(string 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 (or, in Python, a color word).

AddTextDisplay
C/C++: `enum ErrorCode smolAddTextDisplay(simptr sim, char *item)`
Python: `ErrorCode addTextDisplay(string 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.

4.9 Runtime commands

SetOutputPath
C/C++: `enum ErrorCode smolSetOutputPath(simptr sim, char *path)`
Python: `ErrorCode setOutputPath(string path)`
Sets the file path for text output files to `path`.

AddOutputFile
C/C++: `enum ErrorCode smolAddOutputFile(simptr sim, char *filename, int suffix, int append)`
Python: `ErrorCode addOutputFile(string 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.

AddOutputData

C/C++: `enum ErrorCode smolAddOutputData(simptr sim, char *dataname)`
Python: `ErrorCode addOutputData(string dataname)`
Declares the data table called `dataname`, enabling output into it by one or more runtime commands. Spaces are not permitted in the data name.

OpenOutputFiles

C/C++: `enum ErrorCode smolOpenOutputFiles(simptr sim, int overwrite = 0)`
Opens output files for writing. Enter `overwrite` as 1 if any existing file should be overwritten. If `overwrite` is 0 and a file with this name already exists, then Smoldyn asks the user if it should be overwritten. If the user replies no, then this function ends with an error of `ECerror`.

AddCommand

C/C++: `enum ErrorCode smolAddCommand(simptr sim, char type, double on, double off, double step, double multiplier, char *commandstring)`
Python: `ErrorCode addCommand(string type, float on, float off, float step, float multiplier, string 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	-

AddCommandFromString

C/C++: `enum ErrorCode smolAddCommandFromString(simptr sim, char *string)`
Python: `ErrorCode addCommandFromString(str 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.

getOutputData

C/C++: `enum ErrorCode smolGetOutputData(simptr sim,char *dataname,int *nrow,int *ncol,char *array,int erase)`
Python: vector¡vector¡double¿¿ getOutputData(str dataname, bool erase)
Returns data that have been recorded by an observation command (e.g. molcount). Send in the name of the data in `dataname` and pointers to variables that will receive the data in: `nrow`, for the number

of rows, `ncol`, for the number of columns, and `array`, for the data themselves. The data are copied over in this function from the original into the array that is returned, with the result that the data in the array can be modified as desired. *The array needs to be freed by the host code.* All values in this data table are doubles, which is appropriate for some things but not so good for things like species names and molecule states. The array represents a 2D table as a single vector so to read the item at row `i` and column `j`, use `array[i*ncol+j]`. Set `erase` to 1 for the original data to be cleared after it is copied over.

4.10 Molecules

AddSpecies

Python: *sim*.`addSpecies(name: str, state: str = "soln", color: T.Color = "", difc: float = 0.0, display_size: int = 2, mol_list: str = ""`
Python: `ErrorCode addSpecies(str species, str mollist)`
C/C++: `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.

GetSpeciesIndex

C/C++: `int smolGetSpeciesIndex(simptr sim, char *species)`
C/C++: `int smolGetSpeciesIndexNT(simptr sim, char *species)`
Python: `int getSpeciesIndex(str species)`
Returns the species index that corresponds to the species named `species`. Upon failure, this function returns an error code cast as an integer. The "NT" version is identical, but doesn't throw exceptions or print errors to stderr.

GetSpeciesName

C/C++: `char* smolGetSpeciesName(simptr sim, int speciesindex, char *species)`
Python: `str getSpeciesName(int speciesindex, str 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`.

SetSpeciesMobility

C/C++: `enum ErrorCode smolSetSpeciesMobility(simptr sim, char *species, enum MolecState state, double difc, double *drift, double *difmatrix)`
Python: `ErrorCode setSpeciesMobility(str species, MolecState state, float difc, List[float] drift, List[float] difmatrix)`
Python: *species*`.difc`
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.

The last version shown can also be used to get the diffusion coefficient for a species.

AddMolList

C/C++: `int smolAddMolList(simptr sim, char *mollist)`
Python: `int addMolList(str mollist)`
Adds a new molecule list, named `mollist`, to the system.

GetMolListIndex

C/C++: `int smolGetMolListIndex(simptr sim, char *mollist)`
C/C++: `int smolGetMolListIndexNT(simptr sim, char *mollist)`

Python: `int getMolListIndex(str mollist)`
Returns the list index that corresponds to the list named `mollist`. The "NT" version is identical but doesn't throw exceptions or print errors to the stderr output.

GetMolListName
C/C++: `char* smolGetMolListName(simptr sim, int mollistindex, char *mollist)`
Python: `str getMolListName(int mollistindex, str 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`.

SetMolList
C/C++: `enum ErrorCode smolSetMolList(simptr sim, char *species, enum MolecState state, char *mollist)`
Python: `ErrorCode setMolList(str species, MolecState state, str mollist)`
Python: *species*`.mol_list`
Sets the molecule list for species `species` (which may be "all") and state `state` (which may be `MSall`) to molecule list `mollist`.

The last version can either set or retrieve the molecule list.

SetMaxMolecules
C/C++: `smolSetMaxMolecules(simptr sim, int maxmolecules)`
Python: `setMaxMolecules(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.

AddSolutionMolecules
C/C++: `enum ErrorCode smolAddSolutionMolecules(simptr sim, char *species, int number, double *lowposition, double *highposition)`
Python: `ErrorCode addSolutionMolecules(str species, int number, List[float] lowposition, List[float] highposition)`
Python: *species*`.addToSolution(mol: float, highpos: List[float] = [], lowpos: List[float] = [])`
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.

AddCompartmentMolecules
C/C++: `enum ErrorCode smolAddCompartmentMolecules(simptr sim, char *species, int number, char *compartment)`
Python: `ErrorCode addCompartmentMolecules(str species, int number, str compartment)`
Adds `number` solution state molecules of species `species` to the compartment `compartment`. Molecules are randomly distributed within the compartment.

AddSurfaceMolecules
C/C++: `enum ErrorCode smolAddSurfaceMolecules(simptr sim, int speciesindex, enum MolecState state, int number, int surface, enum PanelShape panelshape, int panel, double *position)`
Python: `ErrorCode addSurfaceMolecules(int speciesindex, MolecState state, int number, int surface, PanelShape panelshape, int panel, List[float] 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`.

GetMoleculeCount

C/C++: `int smolGetMoleculeCount(simptr sim, char *species, enum MolecState state)`
Python: `int getMoleculeCount(str species, 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.

SetMoleculeColor

C/C++: `enum ErrorCode smolSetMoleculeStyle(simptr sim, const char *species, enum MolecState state, double *color)`

SetMoleculeSize

C/C++: `enum ErrorCode smolSetMoleculeStyle(simptr sim, const char *species, enum MolecState state, double size)`

SetMoleculeStyle

C/C++: `enum ErrorCode smolSetMoleculeStyle(simptr sim, const char *species, enum MolecState state, double size, double *color)`
Python: `ErrorCode setMoleculeStyle(str species, MolecState state, float size, List[float] color)`
Python: *species*`.setStyle`
Python: *species*`.color`
Python: *species*`.size`
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 (or, in Python, a color word).

4.11 Surfaces

Boundaries

Python: `Boundaries(low: List[float], high: List[float], types: List[str] = field(default_factory=lambda: ["r"]), dim: field(init=False) = 0)`
Python: `setBounds()`
This functionality is only available in Python. There is a class called `Boundaries` and a function called `setBounds`. They do basically the same thing.

SetBoundaryType

C/C++: `enum ErrorCode smolSetBoundaryType(simptr sim, int dimension, int highside, char type)`
Python: `ErrorCode setBoundaryType(int dimension, int highside, str 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.

AddSurface

C/C++: `int smolAddSurface(simptr sim, char *surface)`
Python: `int addSurface(str surface)`
Adds a surface called `surface` to the system.

GetSurfaceIndex

C/C++: `int smolGetSurfaceIndex(simptr sim, char *surface)`

C/C++: `int smolGetSurfaceIndexNT(simptr sim, char *surface)`
Python: `int getSurfaceIndex(str 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. The "NT" version is identical but errors aren't printed to the stderr output and don't throw exceptions.

GetSurfaceName
C/C++: `char* smolGetSurfaceName(simptr sim, int surfaceindex, char *surface)`
Python: `str getSurfaceName(int surfaceindex, str surface)`
Returns the surface name for surface number `surfaceindex` both directly and in the `surface` string. On failure, this returns `NULL`.

SetSurfaceAction
C/C++: `enum ErrorCode smolSetSurfaceAction(simptr sim, char *surface, enum PanelFace face, char *species, enum MolecState state, enum SrfAction action, char *newspecies)`
Python: `ErrorCode setSurfaceAction(str surface, PanelFace face, str species, MolecState state, SrfAction action)`
Python: *surface*`.addAction(face, species: Union[Species, str], action: str, new_spec=None)`
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`. Enter `newspecies` to the name of a new species if the molecule should change species, or as either `NULL` or an empty string if it should not change species.

SetSurfaceRate
C/C++: `enum ErrorCode smolSetSurfaceRate(simptr sim, char *surface, char *species, enum MolecState state, enum MolecState state1, enum MolecState state2, double rate, char *newspecies, int isinternal)`
Python: `ErrorCode setSurfaceRate(str surface, str species, MolecState state, MolecState state1, MolecState state2, float rate, str 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`	
collision from solution state	soln	soln	soln	*reflect*
	"	"	bsoln	transmit
	"	"	bound	adsorb
	"	bsoln	soln	transmit
	"	"	bsoln	*reflect*
	"	"	bound	adsorb
action from bound state	"	bound	soln	desorb
	"	"	bsoln	desorb
	"	"	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*

AddPanel

C/C++: `int smolAddPanel(simptr sim, char *surface, enum PanelShape panelshape, char *panel, char *axisstring, double *params)`
Python: `int addPanel(str surface, PanelShape panelshape, str panel, str axisstring, List[float] 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.

In Python, each panel shape is a separate class. These classes are:

- Rectangle: corner: List[float], dimensions: List[float], axis: str, name=””
- Triangle: vertices: List[List[float]] = [[]], name=””
- Sphere: center: List[float], radius: float, slices: int, stacks: int, name=””
- Hemisphere: center: List[float], radius: float, vector: List[float], slices: int, stacks: int, name: str = ””
- Cylinder: start: List[float], end: List[float], radius: float, slices: int, stacks: int, name=””
- Disk: center: List[float], radius: float, vector: List[float], name=””

GetPanelIndex

C/C++: `int smolGetPanelIndex(simptr sim, char *surface, enum PanelShape *panelshapeptr, char *panel)`
C/C++: `int smolGetPanelIndexNT(simptr sim, char *surface, enum PanelShape *panelshapeptr, char *panel)`
Python: `int getPanelIndex(str surface, PanelShape *panelshapeptr, str 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. The “NT” version is identical but errors aren’t printed to the stderr output and don’t cause exceptions to be thrown.

GetPanelName

C/C++: `char* smolGetPanelName(simptr sim, char *surface, enum PanelShape panelshape, int panelindex, char *panel)`

Python: `str getPanelName(str surface, PanelShape panelshape, int panelindex, str 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`.

SetPanelJump
C/C++: `enum ErrorCode smolSetPanelJump(simptr sim, const char *surface, const char *panel1, enum PanelFace face1, const char *panel2, enum PanelFace face2, int isbidirectional)`
Python: `ErrorCode setPanelJump(str surface, str panel1, PanelFace face1, str panel2, 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.

AddSurfaceUnboundedEmitter
C/C++: `enum ErrorCode smolAddSurfaceUnboundedEmitter(simptr sim, const char *surface, enum PanelFace face, const char *species, double emitamount, double *emitposition)`
Python: `ErrorCode addSurfaceUnboundedEmitter(str surface, PanelFace face, str species, float emitamount, List[float] 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".

SetSurfaceSimParams
C/C++: `enum ErrorCode smolSetSurfaceSimParams(simptr sim, const char *parameter, double value)`
Python: `ErrorCode setSurfaceSimParams(str parameter, float 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.

AddPanelNeighbor
C/C++: `enum ErrorCode smolAddPanelNeighbor(simptr sim, const char *surface1, const char *panel1, const char *surface2, const char *panel2, int reciprocal)`
Python: `ErrorCode addPanelNeighbor(str surface1, str panel1, str surface2, str 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.

SetSurfaceStyle
C/C++: `enum ErrorCode smolSetSurfaceStyle(simptr sim, const char *surface, enum PanelFace face, enum DrawMode mode, double thickness, double *color, int`

`stipplefactor, int stipplepattern, double shininess)`
Python: `ErrorCode setSurfaceStyle(str surface, PanelFace face, DrawMode mode, float thickness, List[float] color, int stipplefactor, int stipplepattern, float shininess)`
Python: *surface*`.setStyle(face, drawmode: str, color: T.Color = "", thickness: float = 1, stipplefactor: int = -1, stipplepattern: int = -1, shininess: int = -1,)`
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 (or, in Python, a color word). `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.

4.12 Compartments

AddCompartment
C/C++: `int smolAddCompartment(simptr sim, char *compartment)`
Python: `int addCompartment(str compartment)`
Adds a compartment called `compartment` to the system.

GetCompartmentIndex
C/C++: `int smolGetCompartmentIndex(simptr sim, char *compartment)`
C/C++: `int smolGetCompartmentIndexNT(simptr sim, char *compartment)`
Python: `int getCompartmentIndex(str compartment)`
Returns the index of the compartment named `compartment`. On failure, this returns an error code cast as an integer. The "NT" version is identical but errors aren't printed to the stderr output or cause exceptions to be thrown.

GetCompartmentName
C/C++: `char* smolGetCompartmentName(simptr sim, int compartmentindex, char *compartment)`
Python: `str getCompartmentName(int compartmentindex, str compartment)`
Returns the name of the compartment that has index `compartmentindex` both directly and in the string `compartment`. Returns `NULL` if an error arises.

AddCompartmentSurface
C/C++: `enum ErrorCode smolAddCompartmentSurface(simptr sim, char *compartment, char *surface)`
Python: `ErrorCode addCompartmentSurface(str compartment, str surface)`
Adds surface `surface` as one of the bounding surfaces of compartment `compartment`.

AddCompartmentPoint
C/C++: `enum ErrorCode smolAddCompartmentPoint(simptr sim, char *compartment, double *point)`
Python: `ErrorCode addCompartmentPoint(str compartment, List[float] point)`
Adds `point` as one of the interior-defining points of compartment `compartment`.

AddCompartmentLogic
C/C++: `enum ErrorCode smolAddCompartmentLogic(simptr sim, char *compartment, enum`

CmptLogic logic, char *compartment2)
Python: ErrorCode addCompartmentLogic(str compartment, CmptLogic logic, str compartment2)
Modifies the current definition of compartment compartment using a logical rule specified in logic and the definition of compartment2.

4.13 Reactions

AddReaction
C/C++: 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)
Python: ErrorCode addReaction(str reaction, str reactant1, MolecState rstate1, str reactant2, MolecState rstate2, int nproduct, List[str] productspecies, List[MolecState] productstates, float rate)
Python: Reaction(subs: List[Species], prds: List[Species], kf, kb=0.0)
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.

GetReactionIndex C/C++: int smolGetReactionIndex(simptr sim, int *orderptr, char *reaction)
C/C++: int smolGetReactionIndexNT(simptr sim, int *orderptr, char *reaction)
Python: int getReactionIndex(List[int] orderptr, str 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. The "NT" version is identical but errors don't get displayed to the stderr output or cause exceptions to be thrown.

GetReactionName
C/C++: char* smolGetReactionName(simptr sim, int order, int reactionindex, char *reaction)
Python: str getReactionName(int order, int reactionindex, str 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.

SetReactionRate
C/C++: enum ErrorCode smolSetReactionRate(simptr sim, int order, char *reaction, double rate, int isinternal)
Python: ErrorCode setReactionRate(int order, str reaction, float 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.

SetReactionRegion
C/C++: enum ErrorCode smolSetReactionRegion(simptr sim, const char *reaction, const char *compartment, const char *surface)

Python: `ErrorCode setReactionRegion(str reaction, str compartment, str 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, "".

SetReactionIntersurface
C/C++: `enum ErrorCode smolSetReactionIntersurface(simptr sim, const char *reaction, int *rulelist)`
Set the intersurface reaction rules for the bimolecular reaction called `reaction`. Intersurface reactions are reactions between two surface-bound molecules that are on two different surfaces. If `rulelist` is `NULL`, then this returns the reaction to the default state, which is that intersurface reactions are not allowed for this reaction. Otherwise, `rulelist` should have one entry for each product. If the entry is 1, then that product is placed on the surface with the first reactant; if it is 2, then that product is placed on the surface with the second reactant. If a reaction has no products, then create a single element in `rulelist` equal to 0 to indicate that intersurface reactions are permitted.

SetReactionProducts
C/C++: `enum ErrorCode smolSetReactionProducts(simptr sim, const char *reaction, enum RevParam method, double parameter, const char *product, double *position)`
Python: `ErrorCode setReactionProducts(str reaction, RevParam method, float parameter, str product, List[float] 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.

4.14 Ports

AddPort
C/C++: `enum ErrorCode smolAddPort(simptr sim, const char *port, const char *surface, enum PanelFace face)`
Python: `ErrorCode addPort(str port, str surface, PanelFace face)`
Python: `Port(name: str, surface: Union[Surface, str], panel: str)`
Adds a port to the simulation. The port will be named `port` and will port at the `face` face of surface `surface`.

GetPortIndex

C/C++: `int smolGetPortIndex(simptr sim, const char *port)`
C/C++: `int smolGetPortIndexNT(simptr sim, const char *port)`
Python: `int getPortIndex(str port)`
Returns the index of the port named `port`. The "NT" version is identical but errors don't get displayed to the stderr output or cause exceptions to be thrown.

GetPortName

C/C++: `char* smolGetPortName(simptr sim, int portindex, char *port)`
Python: `str getPortName(int portindex, str port)`
Returns the name of the port with index `portindex`, both directly and in `port`.

AddPortMolecules

C/C++: `enum ErrorCode smolAddPortMolecules(simptr sim, const char *port, int nmolec, const char *species, double **positions)`
Python: `ErrorCode addPortMolecules(str port, int nmolec, str species, List[float] 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.

GetPortMolecules

C/C++: `int smolGetPortMolecules(simptr sim, const char *port, const char *species, enum MolecState state, int remove)`
Python: `int getPortMolecules(str port, str species, 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.

4.15 Lattices

AddLattice

C/C++: `enum ErrorCode smolAddLattice(simptr sim,const char *lattice,const double *min,const double *max,const double *dx,const char *btype)`
Python: `ErrorCode AddLattice(str lattice,List[float] min,List[float] max,List[float] dx,str btype)`
Adds a lattice to the simulation for hybrid operation. The lattice is named `lattice` and extends from `min` to `max`, with lattice spacing `dx`. The boundary types are given with `btype`, which is a string of either 'r' characters for reflective boundary or 'p' characters for periodic boundary, with one character for each dimension and the dimensions listed in order (e.g. "rrp" is reflective on the x and y axes and periodic on the z axis).

GetLatticeIndex

C/C++: `int smolGetLatticeIndex(simptr sim,const char *lattice)`
C/C++: `int smolGetLatticeIndexNT(simptr sim,const char *lattice)`
Python: `int GetLatticeIndex(str lattice)`
Returns the index of the lattice named `lattice`. The "NT" version is identical but errors don't get displayed to the stderr output or cause exceptions to be thrown.

GetLatticeName

C/C++: `char *smolGetLatticeName(simptr sim,int latticeindex,char *lattice)`

Python: `str getLatticeName(int latticeindex, str lattice)`
Returns the name of the lattice with index `latticeindex`, both directly and in `lattice`.

AddLatticeMolecules
C/C++: `enum ErrorCode smolAddLatticeMolecules(simptr sim,const char *lattice, const char *species,int number,double *lowposition,double *highposition)`
Python: `ErrorCode AddLatticeMolecules(str lattice, str species, int number, List[float] lowposition, List[float] highposition)`
Adds `number` molecules of species `species` to the lattice named `lattice`, randomly positioned over the volume extending from `lowposition` to `highposition`. These molecules will all have state `MSsoln`.

AddLatticePort
C/C++: `enum ErrorCode smolAddLatticePort(simptr sim, const char *lattice, const char *port)`
Python: `ErrorCode AddLatticePort(str lattice, str port)`
Connects port `port` with lattice `lattice`, so that molecules can transition across this port between the particle-based simulation region and the lattice-based simulation region.

AddLatticeSpecies
C/C++: `enum ErrorCode smolAddLatticeSpecies(simptr sim,const char *lattice, const char *species)`
Python: `ErrorCode AddLatticeSpecies(str lattice, str species)`
Not all particle-based Smoldyn species are necessarily listed in a lattice portion of space, so use this function to add species to the lattice region of space. Clearly, the lattice is `lattice` and the species being added is `species`.

AddLatticeReaction
C/C++: `enum ErrorCode smolAddLatticeReaction(simptr sim,const char *lattice, const char *reaction, const int move)`
Python: `ErrorCode AddLatticeReaction(str lattice, str reaction, int move)`
Not all particle-based Smoldyn reactions are necessarily listed in a lattice portion of space, so use this function to add reactions to the lattice region of space. Clearly, the lattice is `lattice` and the reaction being added is `reaction`.