

SGNSim Simulator Manual

Jason Lloyd-Price and Andre S. Ribeiro

July 21, 2009

1 Introduction

To simulate the systems described by SGNSim, we need a chemical simulation capable of operating with some molecular species at extremely low levels. For this purpose, SGNSim includes an optimized implementation of the Stochastic Simulation Algorithm[1]. This document explains how to work with the simulator, and all the functionality it provides.

2 Input

There are several ways a reaction system can be input into the simulator. Most commonly, the reactions will be listed in a file and fed into the program.

Regardless of the source, the reaction system is specified by a series of *identifier-data* pairs. The *identifier* determines how to interpret the *data*. Each section of this document defines its own set of identifiers and the format of their data to communicate the necessary information. For example, to set the stop time of the simulation to 500, one would use the identifier “`stop_time`” (see section 5) with data “500”.

2.1 Input From a File

2.1.1 Format

Information specified in a file follows the following formats:

1. *identifier data*;
2. *identifier* { (*data*;) * }
3. *identifier* !{ *data* }!

The first format is generally used to give simple information, such as the start and stop times of the simulation, or the initial random seed. To input the above example of setting the stop time of the simulation to 500, one could add a line to the file reading:

```
stop_time 500;
```

The second format is used to input a block of similar data, and is equivalent to repeating the first form many times. Lists of reactions and initial populations of molecules can then be added quickly. The following example demonstrates how a set of initial population declarations (see section 7) for a Toggle Switch system can be condensed into a single block:

Format 1	Format 2
population A = 0; population B = 0; population ProA = 1; population ProB = 1;	population { A = 0; B = 0; ProA = 1; ProB = 1; }

The third and final format is designed to allow blocks of data which may contain semicolons or braces inside them to be input. This is primarily useful for writing a chunk of Lua code (see section 10).

2.1.2 Comments

Comments can be placed in an input file to allow inline documentation of the reaction system. Comments follow the C++ comment format, where `//` can be used to comment a line, and `/* ... */` can be used to comment a block of text.

2.1.3 Including an Input File

The identifier `include` can be used to recursively read another input file. The data should be a string containing the filename to include, and nothing else. If the path is a relative path, the file will be searched for in the folder the including file is in, or from the working directory if it is included from the command line.

`#include` can also be used to include a file from within another file, and is an exception to the formatting rules listed above to be compatible with the C preprocessor. Unlike `include`, the filename should be enclosed in double quotes (`"`).

2.2 Input From The Command Line

Identifier-data pairs can also be input from the command line in the format `--identifier data`. This is equivalent to `identifier data`; in a file unless the *data* argument also begins with `--`, in which case it is treated as though *data* was empty. This is mostly useful for including a file containing the reaction system.

Note: If the data contains spaces, quotes should be placed around it so that the operating system sends it as a single argument.

3 Running the Program

The program is generally executed from the command line as:

```
sgns --include filename
```

The included file should contain all the information needed to set up and run the reaction system.

3.1 Progress

If a simulation is expected to take a long time, feedback about its progress can be presented by adding the `--progress` switch to the command line. This will cause the program to output its progress to the console every time the system is sampled (see section 5).

3.2 Warnings

Generally, the warnings given by the program indicate potential problems in the input. If a warning is generated for input that is correct, the warnings may be disabled by adding `--warn off` to the command line.

4 Species Names

Species names follow the C identifier rules. They must begin with a letter or an underscore, followed by a string of letters, numbers or underscores. That is, they must satisfy the regular expression `[a-zA-Z_][a-zA-Z0-9_]*`.

4.1 Multimers

Often, molecules will bind to one another to form complexes. Names of complexes can be formed by concatenating the names of the elementary species with ‘.’. Currently, the complexes are treated internally as a completely separate species. That is, A, B and A.B are completely separate species and will be reported as such, rather than have A.B contribute to the count of the number of As and Bs.

4.2 Compartments

The original SSA algorithm can only correctly model reactions within a well stirred environment. However, many systems are highly compartmentalized and spatial effects such as diffusion times and separation by membranes will greatly affect the dynamics. SGNSim can help model these factors by providing a naming system for molecules in compartments.

A compartment name can be appended to the species name by using @. Compartment names can be a series of numbers, letters or underscores. Reactions moving species among the compartments must still be created manually (ie. written in the input file).

Note: Just as complexes are treated as completely separate species, the same species in different compartments will be treated as separate species as well. That is, A@nucleus, A@cytoplasm and A@1 are distinct species.

5 Simulation Parameters

General simulation parameters are controlled by the following identifiers:

Identifier	Data	Description
<code>time</code>	<code>starttime</code>	Set the initial clock time.
<code>stop_time</code>	<code>stoptime</code>	Tell the simulation to stop when the time reaches <code>stoptime</code> .
<code>seed</code>	<code>[seed]</code>	Set the seed of the random number generator. If the data is empty, it will be initialized based on the system time and number of clock ticks since the program’s start.
<code>readout_interval</code>	<code>[interval]</code>	Set the interval between samples of the system in the output file. If this value is 0, the system will be sampled after every step (reaction or wait list). If the data is empty, output is disabled.
<code>results_file</code>	<code>filename</code>	Set the output filename. The file is not opened until the simulation begins.

By default, the simulation starts and ends at time 0, so `stop_time` must be specified to get any output. The seed is initially randomized as though “`seed;`” appeared at the beginning of the input. The default readout interval is 1 and the default output file is “`results.out`”.

SGNSim uses the Mersenne Twister pseudorandom number generator[2]. This generator has a period of $2^{19937} - 1$.

Note: Data for the `seed` identifier cannot be Lua code.

6 Snapshots

If the system is expected to take a very long time, the simulator can be instructed to output files containing the state of the system at regular intervals. These snapshot files are written in a format that allows the simulator to pick up *exactly* where it left off. The identifiers controlling snapshots are:

Identifier	Data	Description
<code>save_file</code>	<i>filename</i>	Set the snapshot filename.
<code>save_interval</code>	[<i>interval</i>]	Set the interval between snapshots of the system. If the data is empty, snapshots will be disabled.
<code>save_index</code>	<i>index</i>	Set the index to be inserted into the filename of the next snapshot.
<code>save_now</code>	<i>filename</i>	Save a snapshot of the system as it has been read so far. Mostly useful for debugging.

The snapshot filename should contain ‘%’ where the snapshot index should be inserted. Every time a snapshot is taken, the snapshot index is incremented, allowing a succession of snapshots to be written to different files. To restart the simulation at a snapshot, simply `include` the snapshot file.

7 Initial Populations

The initial populations of species are given by the `population` identifier. Populations are usually specified as ‘*speciesname = population*’. Possible uses of the `population` identifier are illustrated below:

Data	Description
<code>A = 5</code>	Set the initial concentration of A to 5.
<code>A += 3</code>	Add 3 to the previously declared initial population of A.
<code>A -= 2</code>	Subtract 2 from the previously declared initial population of A. Initial populations cannot be negative.
<code>A</code>	Equivalent to <code>A += 0</code> .

If a species name is encountered that has not yet been seen before in the input, its initial concentration is initialized to 0. This applies to species in reactions and the waiting list as well as the `+=` and `-=` forms of `population`.

By default, all species are output in the readout file. This can produce a lot of data that may not be necessary. Readout of a species can be explicitly enabled or disabled by placing a `!` or `#` before the species name, respectively.

Readout of newly-discovered species, from any identifier that adds molecular species (`population`, `reaction` and `queue`) can be enabled or disabled by using the `molecule_readout` identifier with data `show` or `hide`.

8 Reactions

Reactions are perhaps the most important aspect of the simulation, and where most of the time is spent when building a model. Reactions are input with the `reaction` identifier in an intuitive and human-readable format:

substrate-list --[*rate-constant*]-> *product-list*

For example, the reaction $A + B \rightarrow C$ with a rate constant of 2 can be input as:

`A + B --[2]-> C`

Zero-order reactions (no substrates) and decay reactions (no products) can be input by simply omitting the *substrate-list* or the *product-list*. For example:

`--[2]-> A`

`A --[2]->`

If more than one of a substrate is consumed in the reaction, such as the reaction $2Y \rightarrow Z$, then the number should be prepended to the substrate name, such as:

```
2Y --[2]--> Z
```

Additionally, the reaction rate is Lua-enabled (see section 10), allowing any formula that can be interpreted by Lua to be inserted, such as:

```
A + B --[math.sqrt(5) + 2 + math.random(4)]--> C
```

This could be used to adjust the reactions' rate constants to variations in the system properties, e.g. volume or temperature, . Lua can also be inserted to calculate the number of reactants consumed or products produced by a reaction by placing the Lua code in square brackets before the species name. This is mostly useful to set these values in global parameters. For example, the reaction

```
A + B --[c.react]--> [n]C
```

allows easy manipulation of both the reaction's rate constant and number of C's produced from a central Lua block.

8.1 Delayed Reactions

By default, products of a reaction are released immediately. In several cases, however, it has been shown that the reaction is better modelled by delaying the release by a certain amount of time. The delay for a given product is specified by placing the delay time in parentheses after the product. This can be used to efficiently simulate multi-step reactions without including all intermediate reactions. For example, the time it takes for a gene to be transcribed by an RNA Polymerase, spliced, translated by a Ribosome, and folded can be simplified into:

```
Promoter + RNAP --[k]--> Promoter(2) + RNAP(20) + P(2000)
```

In this example, the time it takes for the `RNAP` to transcribe the entire gene and be available to bind to another gene is 20 seconds, while it only takes 2 seconds for the gene's `Promoter` to be released so other `RNAP`s can bind and begin transcribing. The end product of the reaction is a protein, and is released long after the `RNAP` finishes transcribing the gene to account for the time it needs to be translated and fold.

If nothing but a number is specified as the delay, the time delay will be assumed to be constant. However, many delays are modelled best when sampled from a distribution, especially when modelling multi-step reactions. For example, when transcribing a gene, RNA Polymerase molecules do not always move at a constant speed across the gene, resulting in a distribution of time delays for the release of the proteins. In this case, the time delay for a given product can be generated from several available distributions. The distribution is specified in the form (*distribution:parameters*). The distribution can be any of the following:

Name	Parameters	Description
<code>delta, const</code>	d	A constant delay of d
<code>gaussian, gaus, normal</code>	μ, σ	A normal distribution with mean μ and standard deviation σ
<code>exponential, exp</code>	λ	An exponential distribution with a mean of $1/\lambda$
<code>gamma</code>	<i>shape</i> and <i>scale</i>	A Gamma distribution with the given shape and scale

For example, the reaction $A + B \rightarrow C$ where C 's release delay follows a Gaussian distribution with mean 10 and standard deviation 2 would be input as:

```
A + B --[k]--> C(gaussian:10,2)
```

Additionally, distributions that allow negative delays to be generated (such as the `gaussian` distribution) will have their negative part truncated, so that negative delays are impossible.

8.2 Virtual Substrates and Reaction Rates

Sometimes, especially when building a reaction system from a system of ODEs, the propensities of the reactions must vary by some non-linear function of the populations of some other species. For example, when the production of one species is suppressed by another.

SGNSim calculates a reaction’s propensity from the following formula:

$$a_i = c_i \prod_{s \in S_i} r_s([s]) \quad (1)$$

Where c_i is the reaction’s stochastic rate constant, S_i is the set of substrates of the reaction, and r_s is the substrate’s rate function (described below). That is, the propensity of the reaction is the product of the reaction’s constant rate and all the rates calculated from the substrates’ rate functions.

A rate function can be assigned to a substrate in a manner similar to assigning a delay distribution to a product. It is specified in the form (*function:parameters*). Unlike delay distributions, there is no default rate function if no *function* is given. If a parameter is omitted, it is assumed to be 1. Available rate functions (where a and b are the parameters, and X is the population of the substrate) are:

Name	Function
gilh	$\prod_{j=1}^a \frac{X-j+1}{j}$
const	a
linear	aX
square, sqr	aX^2
cube	aX^3
pow	bX^a
hill	$\frac{X^b}{a^b + X^b}$
invhill	$\frac{a^b}{a^b + X^b}$
max	$b \cdot \max(a, X)$
min	$b \cdot \min(a, X)$

If no special rate function is given, substrates are assumed to have the **gilh** rate where a is the number of molecules of the substrate that are consumed. **gilh** corresponds to the calculation of H given on p. 5 of [1].

Sometimes, it is necessary to not have any molecules of a given substrate consumed by a reaction. In this case, the substrate becomes a catalyst or inhibitor of the reaction without itself being affected by it, and becomes a ‘virtual’ substrate. This scenario can be input either by specifying 0 as the number of molecules consumed, or by placing a ***** before the species name. Virtual substrates that are not explicitly given a rate are assumed to have the **linear** rate function. For example, the ODEs representing a toggle switch with cooperative binding,

$$\frac{d[A]}{dt} = \frac{k_p}{1 + [B]^c} - k_d[A] \quad (2)$$

$$\frac{d[B]}{dt} = \frac{k_p}{1 + [A]^c} - k_d[B] \quad (3)$$

can be represented with the following reactions:

```
*B(invhill:1,c) --[k_p]--> A
*A(invhill:1,c) --[k_p]--> B
A --[k_d]-->
B --[k_d]-->
```

While the ability to override the usual rate functions greatly enhances the flexibility of the simulator, it is necessary to use them with caution so that populations do not become negative. In this case, the simulation behavior is undefined.

9 The Waiting List

When a reaction with delayed products occurs, the products are placed on a “waiting list”, a priority queue of molecules to be released some time moment later. Initially this list is empty, since the simulation is given no information about what happened in the moments immediately before the simulation began. However, the immediate history of the simulation may turn out to be very important. Such a history can be given by adding waiting list events with the `queue` identifier.

Queue expects data in the form “*speciesname(releasetime)*”. To release multiple molecules at once, the count should be prepended to the molecule name, similar to specifying the number of molecules consumed or produced in a reaction. For example, to release 20 As into the system at $t = 50$, input:

```
queue 20A(50)
```

The `queue` identifier can also be useful to introduce changes to the system mid-simulation. For example, a simulation of a toggle switch can be perturbed at certain points to force a switch to the other stable state (see appendix).

Additionally, similar to the `reaction` format, the number of molecules released at a given time can also be specified with Lua code placed in square brackets before the species name, allowing many variables to be controlled from a central Lua chunk. For example:

```
queue [EquilibriumPoint*2]A(PerturbationTime)
```

Unlike `reaction`, populations in the wait list can be negative, however this should be used very carefully as it could lead to negative populations.

10 Lua

Most numeric fields allow for Lua code to be inserted. Lua code encountered where a number is expected is compiled as though “`return`” was prepended to it. All Lua code is compiled and run immediately when it is encountered, in the order it appears.

A block of generic Lua code can be executed using the `lua` identifier. It is recommended that the data block be delimited with `!{ ... }!` so that semicolons and braces can be used inside the code block without having the block split into several chunks and compiled separately.

A global function “`parse`” is also provided to feed a string from Lua into the parsing system. The function takes one or two parameters. When two parameters are given, the first is interpreted as a string containing the *identifier*, and the second is the *data*. A single string passed to the function will be parsed as though it was from a file.

Documentation for the Lua 5.1 language can be found at <http://www.lua.org/manual/5.1/>.

11 Output

The simulation will produce two sets of output files. When readout is enabled, a tab-delimited spreadsheet in text format will be output in the file specified by the `output_file` identifier. Each row corresponds to a sampling of the system at a given time and contains the sample time, number of reactions completed, number of elements in the waiting list and the populations of all the molecular species at that time. Species are output in the order that they are discovered in the input. If the system is sampled at every iteration of the SSA (`readout_interval` is 0), then an extra column appears indicating the type of step that occurred immediately before that sample.

A block of text can be automatically output at the top of the `output_file` with the `output_file_header` identifier. This can be used to quickly document all output files generated from a certain model. The header is copied verbatim from the data beginning at the first non-whitespace character and ending at the last non-whitespace character, and an endline is appended.

A A Complete Example

toggleswitch_perturb.sim:

```
// Example input file for SGENSim's stochastic simulator
// Model of an Exclusive Toggle Switch subject to perturbations

time 0;
stop_time 1000;

readout_interval 1;
output_file results_toggleswitch.txt;

save_interval 250;
save_index 1;
save_file save_toggleswitch_%.sim;

lua !{
    // Cooperativity of the repression
    coop = 2;

    -- Point where a species would reach
    -- equilibrium with no repression
    equil = 100;
}!

population {
    A = 0;
    B = 0;
}

reaction {
    // Production
    *B(invhill:equil/10,coop) --[equil]--> A;
    *A(invhill:equil/10,coop) --[equil]--> B;

    // Decay
    A --[1]--> ;
    B --[1]--> ;
}

// Perturb the system at t = 500 by releasing lots of As
queue [2*equil]A(250);
// Perturb again at t = 750 by releasing lots of Bs
// We should now see at least one toggling
queue [2*equil]B(500);
// Perturb again at t = 750 by releasing another lots of As
queue [2*equil]A(750);
```

This example can be run by running:

```
sgns --include toggleswitch_perturb.sim
```

This will produce five output files, four of which are snapshots of the system every 250 time units. The time series can be found in `results_toggleswitch.txt` as a tab-delimited spreadsheet.

B Runtime Complexity

The algorithm's best-case runtime is $\Omega(R \cdot \log R)$ and worst-case runtime is $O(R^2)$, where R is the number of reactions in the system. The worst case occurs when the most commonly used substrate is found in $O(R)$ reactions.

References

- [1] D.T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*
- [2] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.