User's Guide for STOCHKIT*

Yang Cao Andrew Hall Hong Li Sotiria Lampoudi
Linda Petzold
Department of Computer Science
University of Calfornia, Santa Barbara

Abstract

Traditional ordinary differential equation-based approaches to simulation of chemical reacting systems fail to capture the randomness inherent in such systems at scales common in intracellular biochemical processes. A number of stochastic algorithms have been proposed and implemented on an ad-hoc basis, but no standard stochastic chemical simulation package exists. We present STOCHKIT, an efficient, extensible stochastic simulation framework developed in the `C++` language that aims to make stochastic simulation accessible to practicing biologists and chemists, while remaining open to extension via new stochastic and multiscale algorithms. STOCHKIT1.0 has the basic simulation ability using the popular Gillespie's SSA algorithm, optimized SSA algorithm, explicit, implicit, and trapezoidal tau-leaping methods. Useful tools are provided to make stochastic simulation more convenient. We provide a Java Converter to convert an SBML file specifying the chemical mechanism to the input files needed for our software. We also provide some basic tools to solve a question of great concern to developers of accelerated stochastic algorithms—how can we verify the accuracy of a stochastic solver, given the inherently random nature of stochastic simulation? The Kolmogorov distance and histogram distance for quantifying differences in statistical distribution shapes are provided in the MATLAB language. For those who need to run the Monte Carlo simulations a large number of times to collect the ensemble, we also provide a convenient MPI interface enabling the Monte Carlo simulation to run on a parallel cluster. This manual presents a detailed introduction to the software.

*This work was supported by the California Institute of Technology under DARPA Award No. F30602-01-2-0558, by the U. S. Department of Energy under DOE award No. DE-FG02-04ER25621, by the National Science Foundation under NSF awards CCF-0326576, CCF-0928912 and ACI00-86061, and by the Institute for Collaborative Biotechnologies through grant DAAD19-03-D-0004 from the U. S. Army Research Office.

1 Introduction

Chemical reaction systems have historically been simulated using ordinary differential equation (ODE) initial value problem (IVP) methods. Such methods appeal to chemical kinetic theory, using reaction rate constants to characterize the evolution of the system in time as a function of the concentrations of the reactant species. As an example, consider the simple reaction

$$\alpha S_1 + \beta S_2 \xrightarrow{k} \gamma S_3, \tag{1}$$

where S_1, S_2, and S_3 represent chemical species, α, β, and γ are positive integers denoting the stoichiometry of the reaction, and k is an experimentally-determined reaction rate constant expressed in units of concentration per unit time (typically mole $\mathrm{L}^{-1}\mathrm{s}^{-1}$).

Classically, the simulation of such a reaction would proceed by writing down the kinetics equations that characterize the instantaneous rates of change of the concentrations of each species S_n as a function of the reaction rate k, and the instantaneous concentrations of each of the reactant species [23]. For Reaction 1, these so-called *reaction rate equations* are:

$$\begin{aligned} \frac{d[S_1]}{dt} &= -k[S_1]^{\alpha}[S_2]^{\beta} \\ \frac{d[S_2]}{dt} &= -k[S_1]^{\alpha}[S_2]^{\beta} \\ \frac{d[S_3]}{dt} &= k[S_1]^{\alpha}[S_2]^{\beta}, \end{aligned} \tag{2}$$

where $[S_n]$ denotes the instantaneous concentration of species S_n [1]. Equations (2) clearly constitute a system of coupled first-order ODEs. A standard initial value problem is formed by specifying the concentration of each species S_n at some initial time t_0. By applying a numerical integration algorithm such as Euler's method or any of a host of Runge-Kutta or multistep methods to the IVP, the concentration of each species at any future time t can be closely approximated. A substantial body of theory exists for the numerical solution of such IVPs; see for example [5].

1.1 The Continuum Approximation

Two related assumptions are inherent in the use of differential equations for chemical simulation—first, that the reactant populations are large, and second,

[1] The exponents in the reaction rate equations (2) are not always simply the stoichiometric coefficients of the reaction, because there may be intermediate species in a reaction whose existence cannot be inferred from the simplified reaction equation. In general, we can say $d[S_i]/dt = f(k, [S_1], \ldots, [S_N])$, where S_i denotes any of the N species participating in the reaction, and f is some differentiable rational function. See [23] for more on the determination of f.

that the reaction environment is well-stirred. Indeed, these are the only conditions under which it is meaningful to discuss the concentrations of the reactant species. Essentially, these assumptions allow us to approximate the system state as a smooth, real-valued continuum. By extension, it is also assumed that an arbitrary degree of accuracy can be achieved with a numerical ODE solver by choosing a suitably small step size.

In the context of industrial chemical reactors and other large-scale domains, the continuum approximation can be quite effectively employed, and results in no significant loss of accuracy. However, it is at best an abstraction—stoichiometry and thermodynamics dictate that chemical reactions are inherently integer-valued stochastic processes. This distinction becomes increasingly important as the scale[2] of the reaction system decreases.

1.2 Biochemistry and Stochastic Simulation

An important class of systems for which the continuum approximation cannot reasonably be made are those arising in cellular processes. Biochemical reaction systems are often quite complex. Reactants may participate in several different reaction pathways, and may be present in quantities that differ by several orders of magnitude. In many cases, the presence or absence of a single important molecule (i.e., an enzyme) can significantly change the course of the reaction. Gene expression is one scenario for which this is often the case [21, 4, 22]. When reactant populations are this small (perhaps in the ones or tens), it makes little sense to speak of their concentrations, and the standard ODE/IVP numerical simulation techniques are no longer applicable. Clearly an alternative approach is called for.

Because chemical reactions result fundamentally from random collisions of discrete molecules, a more faithful numerical simulation technique might try to explicitly model these molecular interactions. Rather than tracing the evolution of the concentration of each species in time by approximating it as a differentiable function, such methods would instead tally the number of molecules of each species, and rely on discrete-event simulation techniques to evolve the population vector in time.

An important advantage of the discrete approach to modeling chemical reaction systems is that it is capable of capturing the randomness inherent in such processes. Because the populations of important species can be quite small, the evolution of the system state may depend strongly on the relatively unlikely interaction of these molecules with those of other, more abundant species. Such an interaction can result in a cascade of other reactions that significantly alters the system state. In a very real sense, the state of such a system at any time

[2]In this context, scale refers to both the spatial dimension of the reaction vessel and the populations of the reactant species.

$t > t_0$ depends not only on the relative abundance of each of the reactant species at the initial time t_0, but on the exact position, velocity, orientation, etc. of *each molecule* at that time.

For a variety of reasons, this sensitive dependence on initial conditions presents a problem. Quantum mechanics (specifically the Heisenberg Uncertainty Principle) asserts the impossibility of ever measuring the exact position and velocity of a single particle, to say nothing of an entire system of hundreds or even millions of molecules. Furthermore, even if it were theoretically possible to do so, so much initial data would be needed that the simulation would be impractical to use.

Stochastic methods offer a convenient shortcut that allows us to avoid the initial condition dilemma. Rather than employing an exact physical model to track the movement and interactions of each molecule, we can instead use a probabilistic model that estimates the likelihood of occurrence of each reaction pathway as a function of the instantaneous population of each reactant species and a rate constant c, derived from the standard reaction rate constant k[10]. We can then draw samples from these probability distributions using standard Monte Carlo techniques to determine the time of the next reaction event, and update the system state according to the stoichiometry of that reaction. Gillespie's *Stochastic Simulation Algorithm* (SSA) [10, 11] is the seminal work in this direction (for a detailed description of the SSA, see [10, 11]). By performing a large number of realizations (and with a sufficiently good random number generator), such stochastic techniques make it possible to visualize the entire distribution of possible system states at time t.

1.3 Accelerated Stochastic Methods

Unfortunately, the rigorous physical basis for the SSA comes at a high cost in computational run-time, as these algorithms derive their accuracy from the explicit modeling of each reaction event. Even systems with relatively modest reactant populations and reaction rates might generate millions of such events in a short time, and the number of events tends to increase superlinearly as the population size increases. Furthermore, biological reaction networks frequently feature some pathways with comparable forward and reverse rates that are large relative to those of the other pathways. These systems effectively involve multiple time scales, with uninteresting high-frequency oscillations superimposed on a slow overall trend driven by the more important reaction channels. In such cases, the vast majority of the simulated reaction events will consist of forward and reverse firings of these uninteresting channels, resulting in relatively little change to the overall population vector. If we are concerned with the distribution of the population at some final time t_f rather than with the exact sequence of reactions that occurred in a particular realization, much of the computation time is wasted on 'uninteresting' reaction events.

Accelerated stochastic methods are an active area of research in computational biochemistry. These algorithms seek to improve upon the performance of the SSA by enabling larger time steps, sacrificing some accuracy in the name of efficiency. Various methods have been proposed (see [12, 7, 35, 33, 9], for example), each with a different strategy for relieving the computational burden of simulating each reaction event. Such techniques appear quite promising, often decreasing the runtime by several orders of magnitude while maintaining a high degree of accuracy. However, no 'silver bullet' has thus far been discovered. Conditions that will cause each technique to fail outright or rapidly to lose accuracy are known. Accelerated stochastic simulation techniques will likely continue to be an important area of research until a suitably robust and efficient, multiscale algorithm is developed.

1.4 Scope of Work

The work presented herein consists of a suite of software tools for stochastic chemical simulation developed for the C++ [29], Java [30] and Matlab [20] programming languages. The intended audience for the programs can be divided into two distinct, yet equally important groups—those doing research on and development of stochastic simulation methods, and those seeking to employ such methods to further their biological or chemical research.

The primary tool presented is a generalized stochastic simulation package StochKit. This package makes it possible to access a variety of stochastic solver algorithms through a unified interface, such that any available solver can be applied to a given problem by changing a single function parameter. In addition, due to the modular nature of the package, those developing new solver algorithms (or simply refining existing ones) need only supply a new routine that captures their particular innovation (i.e. stepsize selection, single step execution, data management, etc.).

A number of secondary tools are provided to complement the stochastic simulation solvers. The DataAnalyzer assists with collecting and analyzing distribution statistics generated by making ensemble runs of the solver on a given problem. These tools are useful for comparing the accuracy of various solver algorithms, and for visualizing the range of outcomes a particular problem can generate. The SBML2StochKit Converter provides a tool to convert an SBML [14] file to the input files required by StochKit. Using this converter, the user can conveniently construct their problem files using any SBML constructor and run the simulation using StochKit. In many applications, Monte Carlo simulation has to be run a large number of times to collect the ensemble. This work is natural for parallel computing. We provide an MPI interface to the StochKit package. The user can easily use this tool to collect the ensemble using a parallel cluster. Thus, StochKit and its supporting tools should be effective and easy to use for both of its intended audiences.

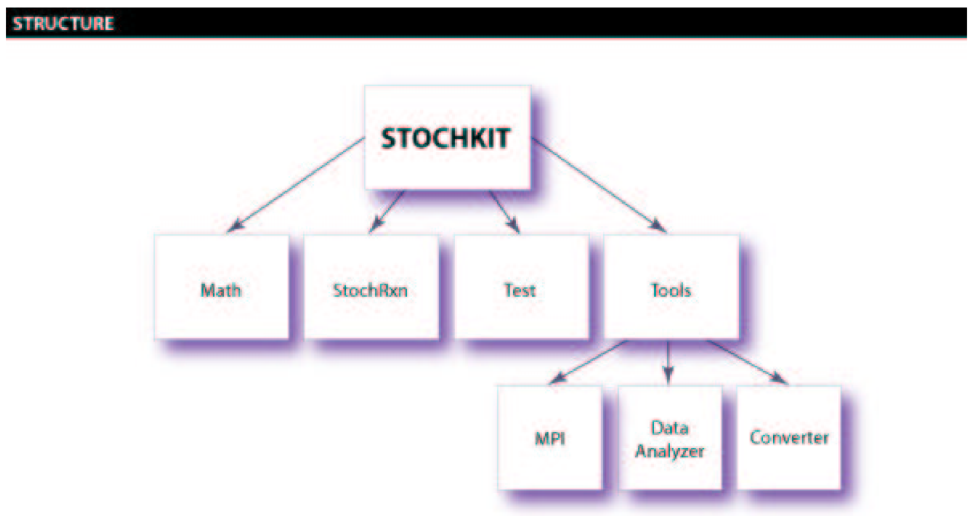

Figure 1: Structure of StochKit

The structure of StochKit is shown as in Figure 1.4. In this manual we will explain the details of each block.

2 Mathematical Primitives in C++

StochKit was initially developed in the Matlab language[3]. The development of the C++ implementation of StochKit was slowed by the low level of support for scientific programming provided by the C++ standard library. Specifically, matrix and vector primitives are not built into the language, though the strong support for data abstraction and operator overloading make it relatively easy to create linear algebra libraries whose syntax and semantics closely match those of standard mathematical notation. A number of commercial ([16]) and open-source ([28, 24, 31, 27, 15]) libraries provide this capability already.

A review of the freely available linear algebra libraries for C++ was inconclusive. LAPACK++ ([15]) seemed like a logical choice due to its affiliation with the successful LAPACK Fortran/C libraries [2], but it has been superseded by TNT ([24]), which appears to be no longer supported. MTL ([28]), Blitz++ ([31]), and TNT ([24]) were rejected for syntactical reasons—although improved performance is the primary goal of the C++ implementation, a secondary goal is to maintain as much syntactic commonality with the Matlab implementation as is feasible. For this reason, it was decided that it would be worthwhile to spend some time implementing a small library whose syntax and semantics conformed to those of Matlab. This decision was made with some trepidation, and will be worth revisiting in the future if a "standard" C++ linear algebra library emerges.

Basic Capabilities

The CSE::Math library defined and implemented in the *Math* directory provides a set of operations to ease the transition from MATLAB to `C++` development. It is not intended to serve as a general-purpose high-performance linear algebra library, but rather to minimize surprises for experienced MATLAB programmers while remaining as efficient as possible within the context of STOCHKIT. The basic capabilities are outlined below.

Vector and (dense) Matrix Classes Basic linear-algebraic quantities are implemented with strict deep-copy. This means that unnecessary memory allocation and data movement may sometimes occur, but guarantees that modifications to one vector or matrix do not inadvertently cause changes in others via aliasing. Vectors and Matrix are implicitly column-oriented. This helps to construct internal calls to the popular linear algebra Fortran packages (LAPACK or LINPACK) when necessary. The `Vector` and `Matrix` classes and all of the following operations are defined in the `CSE::Math` namespace.

Indexing Operations Elements from vectors and matrices can be accessed using the '`()`' operator as in MATLAB. Unlike MATLAB, indices start at 0 (as is customary in `C` and `C++`). Rows and columns of matrices can be selected using the `Row()` and `Col()` member functions. This is a necessary departure from the MATLAB slicing syntax, as the '`:`' operator is not available for overloading in `C++`. These operations return objects that serve as proxies for the row or column of the original matrix, such that the expression '`m.Row(0) = 1;`' fills the top row of the matrix with ones, rather than creating a separate vector object and filling it with ones. However, all operations defined for vectors are also defined for matrix row and column proxies, so they can be treated as vectors when convenient without excessive data copying.

Scalar-Vector and Scalar-Matrix Operations As in MATLAB, binary operators '`+`', '`-`', and '`*`' involving any mix of scalar and vector or scalar and matrix operands are provided. The resulting value is a matrix or vector equal to the original with the scalar operation applied to each element. Thus, the expressions '`y = x * 5;`' and '`y = 5 * x;`' (where `x` and `y` are vectors) are equivalent, with the result that $||\mathtt{y}|| = 5||\mathtt{x}||$. The division operator, '`/`', can have a vector or matrix on the left side and a scalar on the right, and similarly applies the scalar division to each element of the matrix. In addition, the '`+=`', '`-=`', '`*=`', and '`/=`' operators are provided for vector or matrix left sides, and result in in-place modification of the left side operand. Thus, '`x += 5;`' is semantically equivalent to '`x = x + 5;`', but avoids the expensive creation of a temporary object to hold the intermediate result. These operators have no analog in MATLAB, but will be familiar to `C` and `C++` programmers and are provided because they are significantly more efficient when applicable.

Vector-Vector Operations Vectors can be added and subtracted just as in MATLAB. An `InnerProduct()` function is provided to obviate the need for a data-copying transpose operation (although Matrix objects do have a defined `Transpose()` function). The `Norm()` function can compute any vector p-norm. As mentioned above, all vector operations are equally valid for matrix row and column proxies.

Matrix-Vector Operations Vectors can be left-multiplied by matrices, and linear systems can be solved for a particular right side vector using the `SolveGE()` function. If the USELAPACK option is chosen, `SolveGE()` internally calls LAPACK Fortran LULinearSolver dgesv(), otherwise we provide a simple implementation of Gaussian Elimination with partial pivoting. Our numerical test shows that the LAPACK function is 10% slower but should be more robust.

Matrix-Matrix Operations Matrices can be added, subtracted, and multiplied using standard operator syntax. An identity matrix of any size can be constructed via the `Identity()` function, and the `Zeros()` and `Ones()` functions are analogous to the MATLAB routines of the same names.

It was not possible to exactly match MATLAB syntax in all cases, as the set of operators available in `C++` and MATLAB differ slightly. However, in most cases the changes were minor and unsurprising, and translation from MATLAB code to `C++` was a simple task once the CSE::MATH library was in place.

Random Number Generation

High quality pseudorandom number generation is the cornerstone of any stochastic simulation system. In fact, statistical results can only be relied upon if the independence of the samples can be guaranteed. The standard library routines `rand()` from `C`, and `random()`, which is not present on all UNIX systems, could in principle be used to provide uniform random integers for our simulations. However the implementation of these routines is not the same on all systems, and the quality of the various implementations is often poor. Favoring speed over quality, `rand()` is usually implemented using a one-seed linear congruential algorithm [25], and hence produces a sequence with period 2^{32} $(4,294,967,296)$ or less on 32-bit architectures. This short period suggests that repetition of the sequence is a realistic possibility for algorithms that take many steps (e.g. SSA) or when many realizations are needed.

For the `C++` implementation of StochKit, a higher-quality source of pseudorandom numbers was needed. We chose to use the Scalable Parallel Random Number Generators Library (SPRNG) [18, 19]. SPRNG provides multiple generators; by default we use the linear congruential generator, because it is the fastest one, but only one variable needs to be modified in order to switch to a different type of generator. An added bonus of using SPRNG is that we now have the ability to parallelize our code at will, since SPRNG provides a facility for generating parallel streams of random numbers that are guaranteed to be uncorrelated.

Furthermore, non-uniform distributions are needed—the τ-leaping algorithm requires Poisson deviates, and one can easily imagine future stochastic algorithms that require numbers drawn from normal, binomial, or other distributions. The library selected for this purpose was *RANLIB.C* [6], freely available on Netlib [8]. *RANLIB.C* provides generators for a wide variety of distributions based on a common uniform generator with period greater than 2×10^{18} [17], making sequence repetition extremely unlikely. Nevertheless, we adapted *RANLIB.C* to use SPRNG's linear congruential generator as its uniform generator, in order to further minimize the probability of sequence repetition. A simple `C++` wrapper was provided for a few of the *RANLIB.C* routines to improve user-friendliness. These functions are also defined in the `CSE::Math` namespace.

3 Stochastic Simulation Routines

This directory contains the core simulation functions of our package. With possible further extensions in mind, these functions were designed to be as modular as possible. The simulation methods and major drivers are provided as functions. The user can call these functions in his/her own code. Examples are provided in the Test directory. The following is the basic outline followed by all algorithms:

1. Configuration Program inputs are analyzed and used to set key solver parameters (e.g. tolerances, step size, etc.).

2. Solution Loop The following steps are repeated at each time t as the solution advances from initial time t_0 to final time t_f:

   1. Select the step size dt.
      - For SSA, this involves determining the time at which the next reaction will fire.
      - For τ-leaping methods, this involves determining the largest step that can be taken without significantly altering the reaction propensity function (the so-called *leap condition*). In the adaptive mode[32], the selected τ will be further compared with the time stepsize given by SSA to decide which method to use in the next step.

   - For any solver, a check is needed to make sure that $t + dt \leq t_f$.

2. Take a single step to $t + dt$ with the selected algorithm.
   - For SSA, determine which reaction is imminent, and generate a new state vector reflecting the occurrence of this reaction.
   - For τ-leaping methods, determine how many times each reaction channel fired during the interval $[t, t + dt)$, and generate a new state vector reflecting the cumulative effect of these reactions.
   - For any method, checks must be implemented to prevent non-physical behavior. In particular, reactant species populations must be non-negative.
3. Update the solution history.
   - If the user wishes to see the entire solution history, append the new state vector to the history.
   - If the user wishes to see the solution history sampled at some fixed interval, determine if $t + dt$ is one of the sample times, and if so, append the new state vector to the history.
   - If the user cares only about the final state of the solution, over-write the history with the new state vector.
4. Repeat if $t < t_f$.

3. Cleanup & output Terminate the solver loop and return the solution history to the user.

All algorithms are implemented in the CSE::StochRxn library. In order to use this library, we highly recommend users try the test examples provided in the TEST directory. The following are the concepts involved in this library:

System A system contains three necessary elements. The first element is the state vector, which represent the population of all the species in the system. If a system involves N molecular species $\{S_1, \ldots, S_N\}$, the state vector is denoted by $X(t) = (X_1(t), \ldots, X_N(t))$, where $X_i(t)$ is the number of molecules of species S_i in the system at time t. The user is required to provide N, the dimension of the state vector and x_0, the initial value of X. The second element is the reaction set, which contains all the reaction channels. We denote these by $\{R_1, \ldots, R_M\}$. In the current version, the user is required to provided all the possible reaction channels. We are aware that there are some applications in which it is not practical to list all possible channels. That situation is not handled in the current version. The last element is the simulation time interval. The user is required to provide the initial time t_0 and the final time t_f.

ReactionSet A ReactionSet contains all the possible reaction channels in a system. Each reaction channel R_j is characterized by the *propensity function* a_j and by the *state change vector* $\nu_j = (\nu_{1j}, \dots, \nu_{Nj})$: $a_j(x)dt$ gives the probability that one R_j reaction will occur in the next infinitesimal time interval $[t, t + dt)$, and ν_{ij} gives the change in the S_i molecular population induced by one R_j reaction. The user must provide M, the number of reaction channels, an $N \times M$ Matrix ν, the stoichiometric matrix, and a propensity function PropensityFunc, which takes a Vector of states as an argument and returns a Vector of propensities.

SolverOptions For a system, different simulation and output methods can be chosen by setting up SolverOptions. Some important parameters are:

StepControl This is an option to choose different simulation strategies. Two possible options are provided. One is FixedStep (value = 0). Under this option, a fixed simulation method (SSA or one of the tau-leaping methods) will be applied. The other is AdaptiveStep (any nonzero value). Under this option, the non-negative tau-leaping method [32] will be applied, which will automatically switch between SSA and tau-leaping methods according to the algorithm given in [32]. If the user doesn't set this value, the default value will be taken as FixedStep(0).

StepsizeSelectionFunc This is a function which decides the time progress of the current step. The possible options are SSADirect_Stepsize (default), which uses Gillespie's SSA method to decide the next reaction time and Fixed_Stepsize, which uses a fixed stepsize. The second option can be used with leaping methods. If it is chosen, the user should provide an initial stepsize in the parameter *initial_stepsize*. Other options include Gillespie_Stepsize (Gillespie's stepsize selection strategy [12]), GillespiePetzold_Stepsize (Gillespie and Petzold's improved stepsize selection strategy [13]), and Cao_Stepsize (Cao et. al's componentwise stepsize selection formula [34].

SingleStepFunc This is a function to update the states in each step. The possible options are:

SSA_SingleStep, OSSA_SingleStep and MSSA_SingleStep They pick one reaction channel as in the SSA method, and updates the states affected by this reaction; SSA_SingleStep is the default value.

ExplicitTau_SingleStep It updates the states according to the explicit tau-leaping method [12].

ImplicitTau_SingleStep It updates the states according to the implicit tau-leaping method [26].

ImplicitTrapezoidal_SingleStep It updates the states according to the trapezoidal tau-leaping method[1].

Note that if an implicit method is chosen, the user should provide the absolute tolerance *absolute_tol* and relative tolerance *relative_tol* for Newton iteration at each time step. The Jacobian function for the propensity function should also be provided for the linear system in the Newton iteration. The user can choose to provide the analytic function to evaluate the Jacobian, or to use the finite difference method.

Progress_interval This is an option for the simulation progress report. The program will print the states to the standard output at the end of each such progress interval. This can be helpful for debugging purposes. This value gives the length of progress interval by the number of steps between reports. The user should usually select a large number; Otherwise the progress report will slow down the simulation.

StochRxn This is the function to compute one realization. It requires arguments of initial state vector x_0, starting time t_0, ending time t_f, ReactionSet and SolverOptions. Then it performs the simulation according to the setup. After the simulation is done, a simulation history is returned to the function call.

CollectStats This is the function to compute many realizations and generate ensembles. It is useful to study a system's statistical properties. Besides the arguments one single realization requires, this function requires another argument: the ensemble dimension, which represents how many Monte Carlo realizations the user wants to perform. The ensemble data is returned to the function call after all the simulations are done. This ensemble data can be written in a text file. By using the data analysis tools provided, the user can conveniently generate plots from the ensemble data. The current version provides only the ensemble for the states at the final time.

The methods applied in StochKit is shown in Figure 3 Full references for these functions are listed in the Appendix.

4 Solving the Test Problem with StochKit

In this section we will show how to use StochKit to solve the DimerDecay example. A UNIX-style command line interface will be used for the examples.

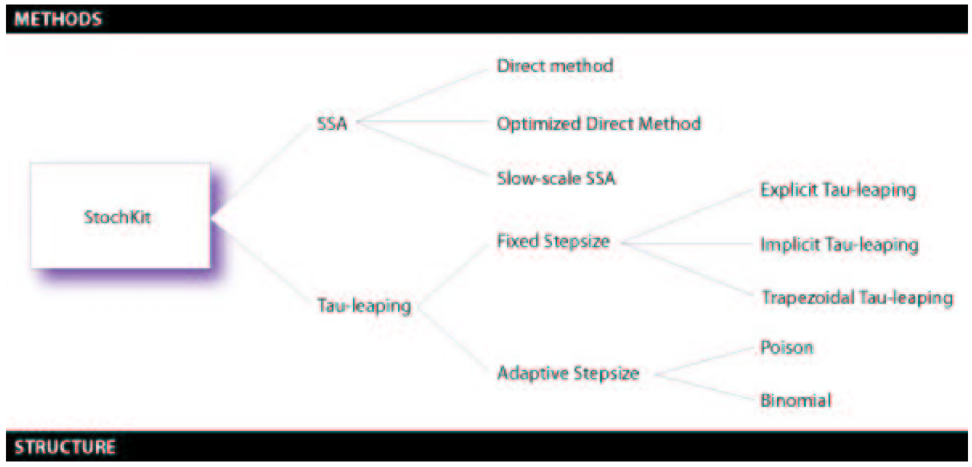

Figure 2: Methods implemented in StochKit

4.1 Preparing for the Run

As introduced above, we need to first encode the test problem in the `C++` language before we can invoke the solver routines. We will need to provide the initial state vector x_0, stoichiometric matrix ν, propensity function and propensity Jacobian functions. The easiest way to do so is to create a ".cpp" problem definition file to hold all of it. We can simplify things by including a header file `ProblemDefinition.h` and defining the functions it declares in our file.

The results of doing this for our test problem are shown in Figure 3. Note that the `InitialState()` and `Stoichiometry()` functions are each called a single time before invoking the solver, and are responsible for generating the vector $\boldsymbol{x}_0$ and the matrix $\boldsymbol{\nu}$. In this example, we take advantage of the zero-initializing `Vector` and `Matrix` constructors in the Matlab emulation library to create `x0` and `nu`, then simply fill in the non-zero elements manually. Equivalently, we could have used the `Zeros()` functions to achieve the same effect.

The `Propensity()` and `PropensityJacobian()` [3] functions have specific interface requirements due to the solver's dependence on them. Specifically, the `Propensity()` function expects to receive a state vector `x` of dimension N (the number of species), and return a propensity vector of dimension M (the number of reaction channels). Likewise, the `PropensityJacobian()` function expects to receive a state vector `x` of dimension N and return a Jacobian matrix of dimension N by M.

With all the details of our problem contained in a single file, we can link it to single-realization and statistical endpoint sampling driver programs at will.

[3]The user does not always need this Jacobian function. The Jacobian is only needed for implicit tau methods to solve stiff problems.

```
#include "ProblemDefinition.h"

Vector InitialState()
{
  Vector x0(3, 0.0);
  x0(0) = 1e5;
  return x0;
}

Matrix Stoichiometry()
{
  Matrix nu(3, 4, 0.0);

  nu(0,0) = −1; nu(0,1) = −2; nu(0,2) = 2;
  nu(1,1) = 1;  nu(1,2) = −1; nu(1,3) = −1;
  nu(2,3) = 1;
  return nu;
}

Vector Propensity(const Vector& x)
{
  const double c1 = 1.0, c2 = 0.002, c3 = 0.5, c4 = 0.04;
  const double x1 = x(0), x2 = x(1);
  Vector a(4);
  a(0) = c1 * x1;
  a(1) = (c2/2.0)*x1*(x1−1);
  a(2) = c3 * x2;
  a(3) = c4 * x2;
  return a;
}

Matrix PropensityJacobian(const Vector& x)
{
  const double c1 = 1.0, c2 = 0.002, c3 = 0.5, c4 = 0.04;
  const double x1 = x(0);
  Matrix j(4, 3, 0.0);
  j(0,0) = c1;
  j(1,0) = c2 * x1 − c2/2.0;
  j(2,1) = c3;
  j(3,1) = c4;
  return j;
}
```

Figure 3: Listing of C++ test problem definition file DimerDecayRxn.cpp.

4.2 Generating a Single Realization

With our test problem in place in file `DimerDecayRxn.cpp`, we need a driver routine to exercise it. In this section we will create a driver routine that can generate a single realization of our test problem using any of the supplied exact or accelerated methods. The driver file appears as Figure 4. It will be helpful to refer to it as we discuss its structure. We will refer to the driver as `SingleDriver.cpp`.

In lines 1–9 of the driver file, we include the necessary declarations from the *stoch_rxn* and CSE::MATH libraries. In particular, the functions defined in `DimerDecayRxn.cpp` are declared in `ProblemDefinition.h` (line 2). Lines 13–26 simply parse the command line and set the solver algorithm selection string (`solverName`) and output file name (`outFile`) accordingly. Lines 28–34 get the initial state vector and stoichiometry matrix by calling the functions defined in `DimerDecayRxn.cpp`. A `ReactionSet` object is prepared in line 31 to collect all the problem-specific information into a format that the solvers understand. Line 36 creates the `SolverOptions` structure using the `ConfigStochRxn()` function. For users who are only interested in an efficient SSA or tau-leaping solver, they do not need to worry about the `SolverOptions`. For SSA, simply set `SolverOptions opt = ConfigStochRxn(1);`. For tau-leaping method that automatically chooses τ value and switch to SSA when necessary, set `SolverOptions opt = ConfigStochRxn(1);`. For users who are also interested in testing different formulas, they can modify these options and get different results. Examples can be found in test directory.

The call to the `StochRxn()` solver appears in line 40 of the code. The call returns a `SolutionHistory` object that contains a single `SolutionPt` object containing time, population vector, and selected stepsize information for each step made by the solver. An auxiliary function, `WriteHistoryFile()` is invoked after the realization completes, to write the reaction history to a text file readable by MATLAB or any other program with plotting capability. Finally, lines 46–51 handle error situations and cleanup.

To invoke the solver we compile both files and link them together and with the `CSE_Math` and `CSE_StochRxn` libraries. Assuming we've named our executable file "`dimersingle`", a transcript of a single run using the implicit τ-leaping solver follows:

```
dimersingle rxnhist.txt
x0 = [  1000  0  0  ]
Nu:
        -1      -2      2       0
        0       1       -1      -1
        0       0       0       1

History written to file <rxnhist.txt>.
```

```
#include "StochRxn.h"
#include "ProblemDefinition.h"
#include "Vector.h"
#include "Matrix.h"
#include <stdlib.h>

using namespace CSE::Math;
using namespace CSE::StochRxn;

int main(int argc, const char* argv[])
{
  try {
    // Parse arguments:
    double tf = 10.0;
    const char *solverName, *outFile;

    if (argc != 4) {
      std::cerr << "Usage:  singletest <tf> "
                << "<solver> <out file>\n";
      exit(EXIT_FAILURE);
    }
    else {

      solverName = argv[2];
      outFile = argv[3];
    }

    // Set up the problem:
    Vector x0 = InitialState();
    Matrix nu = Stoichiometry();
    ReactionSet rxns(nu, Propensity);

    std::cout << "x0 = " << x0 << '\n';
    std::cout << "Nu: \n" << nu << '\n';

    // Configure solver
    SolverOptions opt = ConfigStochRxn(1);

    // Make the run & report results
    SolutionHistory sln = StochRxn(x0, 0, tf, rxns, opt);
    WriteHistoryFile(sln, outFile);

    std::cerr << "History written to file <"
              << outFile << ">.\n";
  }
  catch (const std::exception& ex) {
    std::cerr << "\nCaught " << ex.what() << '\n';
  }

  return 0;
}
```

Figure 4: Listing of `C++` single-realization driver file `SingleDriver.cpp`. The file is compatible with any problem defined according to `ProblemDefinition.h` conventions.

At the completion of this run, `rxnhist.txt` contains the time and population vector data for each step, with one step per line. This file can be opened and plotted with MATLAB.

4.3 Generating an Endpoint Population Distribution

STOCHKIT was really designed with the generation of large numbers of realizations in mind. We can achieve this with a few small modifications to the single realization driver code presented in Figure 4. The distribution generator code appears in Figure 5. We will assume that it is stored in source file `StatDriver.cpp`.

The main differences between the `StatDriver.cpp` and `SingleDriver.cpp` files are highlighted here. In line 1, we include the `CollectStats.h` header file rather than the `StochRxn.h` file. An additional command-line parameter has been added to capture the number of realizations to be performed, resulting in the addition of lines 14 and 24. Then, in lines 42 and 43, the `CollectStats()` solver driver function is invoked, producing an `EndPtStats` object that contains the final population vector for each realization. In line 45, these endpoint population vectors are written to the specified text file via the `WriteStatFile()` auxiliary function, one realization to a line.

Having prepared the distribution generator driver, we compile and link as before. Assuming that we create an executable named `dimerdist`, a 10, 000 realization ensemble run would appear as follows:

```
dimerdist 10000 endpts.txt
x0 = [  1000  0  0  ]
Nu:
        -1      -2      2       0
        0       1       -1      -1
        0       0       0       1

Run 10000 of 10000
Endpoints written to file <endpts.txt>.
```

We will explain in Section 5.1 how to draw plots based on the ensemble data.

4.4 Test Problems

We have provided two simple test problems from the literature. One is the the Dimerdecay problem studied in [12], which consists of three species S_1, S_2 and

```
#include "CollectStats.h"
#include "ProblemDefinition.h"
#include "Vector.h"
#include "Matrix.h"
#include <stdlib.h>

using namespace CSE::Math;
using namespace CSE::StochRxn;

int main(int argc, const char* argv[])
{
  try {
    // Parse arguments:
    int iterations;
    double tf = 10.0;
    const char *solverName, *outFile;

    if (argc != 4) {
      std::cerr << "Usage:  stattest <iterations> "
                << "<tf> <solver> <out file>\n";
      exit(EXIT_FAILURE);
    }
    else {
      iterations = atoi(argv[1]);

      solverName = argv[2];
      outFile = argv[3];
    }

    // Set up the problem:
    Vector x0 = InitialState();
    Matrix nu = Stoichiometry();
    ReactionSet rxns(nu, Propensity);

    std::cout << "x0 = " << x0 << '\n';
    std::cout << "Nu: \n" << nu << '\n';

    // Configure solver
    SolverOptions opt = ConfigStochRxn(1);

    // Make the run & report results
    EndPtStats endpts =
      CollectStats(iterations, x0, 0, tf, rxns, opt);
    WriteStatFile(endpts, outFile);

    std::cerr << "Endpoints written to file <"
              << outFile << ">.\n";
  }
  catch (const std::exception& ex) {
    std::cerr << "\nCaught " << ex.what() << '\n';
  }

  return 0;
}
```

Figure 5: Listing of `C++` multiple-realization driver file `StatDriver.cpp`.

S_3 and four reaction channels:

$$\begin{aligned} S_1 &\xrightarrow{c_1} 0 \\ S_1 + S_1 &\xrightarrow{c_2} S_2 \\ S_2 &\xrightarrow{c_3} S_1 + S_1 \\ S_2 &\xrightarrow{c_4} S_3. \end{aligned} \tag{3}$$

The propensity functions are given by

$$a_1 = c_1 x_1, \quad a_2 = \frac{c_2}{2} x_1(x_1 - 1), \quad a_3 = c_3 x_2, \quad a_4 = c_4 x_2,$$

Following [12], we chose values for the parameters

$$c1 = 1.0, \quad c2 = 0.002, \quad c3 = 0.5, \quad c4 = 0.04.$$

The initial conditions were changed to $x_1(0) = 1000$, $x_2(0) = 0$ and $x_3(0) = 0$, and the problem was solved on the time interval $[0,\ 10]$.

The second example is the Schlögl reaction. This reaction is famous for its bistable distribution. The reactions are given by

$$\begin{aligned} B_1 + 2X &\underset{c_2}{\overset{c_1}{\rightleftharpoons}} 3X, \\ B_2 &\underset{c_4}{\overset{c_3}{\rightleftharpoons}} X, \end{aligned} \tag{4}$$

where B_1 and B_2 denote buffered species whose respective molecular populations N_1 and N_2 are assumed to remain essentially constant over the time interval of interest. Let

$$x(t) = \text{number of } X \text{ molecules in the system at time } t. \tag{5}$$

The state change vectors are $\nu_1 = \nu_3 = 1$, $\nu_2 = \nu_4 = -1$. The propensity functions are

$$\begin{aligned} a_1(x) &= \tfrac{c_1}{2} N_1 x(x-1), \\ a_2(x) &= \tfrac{c_2}{6} x(x-1)(x-2), \\ a_3(x) &= c_3 N_2, \\ a_4(x) &= c_4 x. \end{aligned} \tag{6}$$

For some parameter values, this model has two stable states. The parameter set that we used in our simulation has this property, and was given by

$$\begin{gathered} c_1 = 3 \times 10^{-7}, \quad c_2 = 10^{-4}, \quad c_3 = 10^{-3}, \quad c_4 = 3.5, \\ N_1 = 1 \times 10^5, \quad N_2 = 2 \times 10^5. \end{gathered} \tag{7}$$

We set the simulation interval from $t = 0$, with initial state $x(0) = 250$, to time $T = 4$. Plots of different trajectories are shown in Figure 6

Figure 6: Different trajectories for the Schlögl model.

5 Tools

5.1 Data Analyzer

The data analysis in deterministic simulation usually gives a trajectory of the system states from the initial time to the final time. In stochastic simulation, that can also be done. But of more relevance in the stochastic simulation are the statistical properties. By repeating stochastic simulation for many times with different random number sequences, we obtain an ensemble of the data. The histogram can be plotted from the ensemble which represents the probability density function of the corresponding variable. The cumulative distribution function can also be plotted from the ensemble. In the tools directory, three MATLAB subroutines are provided for plotting distributions.

A question that often arises in the research of stochastic simulation is how to measure the error of the stochastic simulation. One may compare the ensemble from the simulation with the ensemble from the experimental data, or compare it with the ensemble from an "exact" simulation such as SSA. In either way, the answer requires calculation of the distance between two distributions. A number of distribution distances have been proposed in the literature. In this package we provide three functions to calculate two distribution distances. One is the total variance distance, which is related to the histogram, defined as

$$D(X,Y) = \int |p_X(s) - p_Y(s)| ds, \tag{8}$$

where p_X and p_Y are the probability density functions of X and Y respectively. Another is the Kolmogorov distance, which is related to the cumulative distribution function, defined as

$$K(X,Y) = \max_{-\infty<x<\infty} |F_X(x) - F_Y(x)|, \tag{9}$$

where F_X and F_Y are the distribution functions of X and Y respectively [4]. The details of the provided functions are listed below:

histoplot_int is a histogram plot subroutine which takes a vector as the first argument. It assumes that the vector elements are integers. If a real-valued vector is given, that vector will automatically be rounded to integers. The probability for a variable to be at a particular value (integer) is given by the number of samples, whose value is equal to that integer, divided by

[4]Note that the distribution function and the probability density function have the relationship:

$$F(x) = \int_{-\infty}^{x} p(s) ds. \tag{10}$$

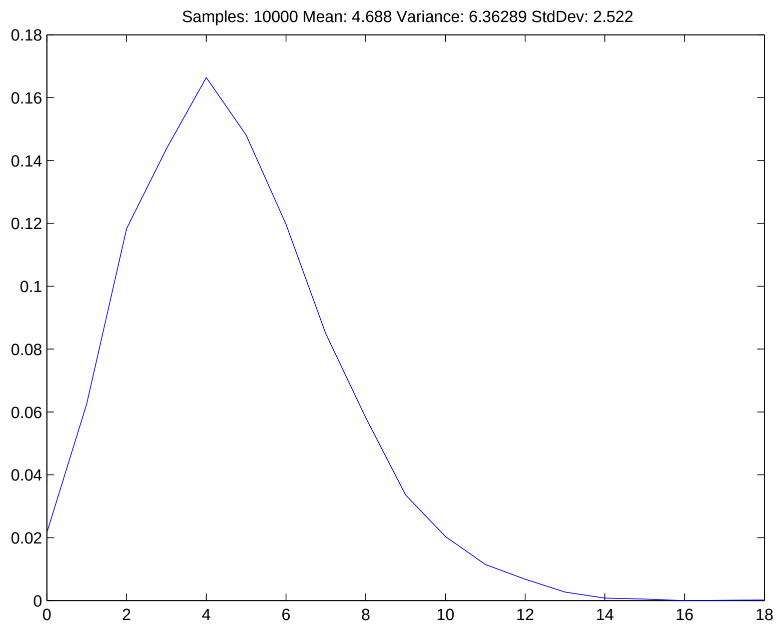

Figure 7: Histoplot_int plot for X_1 in the Dimerdecay example. This plot is based on $10,000$ samples.

the total number of samples. The plot gives the calculated probabilities for all the possible values. The mean, variance and standard deviation are also calculated for the variable.

histoplot_real is a histogram plot subroutine which takes a vector as the first argument, and a positive integer as the second argument. The positive integer K is the number of bins. Since we do not assume that the vector contains only integers, bins are used to measure the histogram. The bins are constructed by dividing the whole interval into K subgroups (bins). The probability that a random variable falls into a bin is the number of samples which are in that bin divided by the total number of samples. The plot gives the calculated probabilities for all the bins. The mean, variance and standard deviation are also calculated for the variable.

cdfplot is a cumulative distribution function (cdf) plot subroutine which takes a vector as the first argument. The cdf value is calculated as the number of samples that are smaller than a particular value, divided by the total number of samples. Here we do not distinguish whether the input vector are integers. The mean, variance and standard variance are also calculated for the variable.

histodistance_int calculates the histogram distance between two random vari-

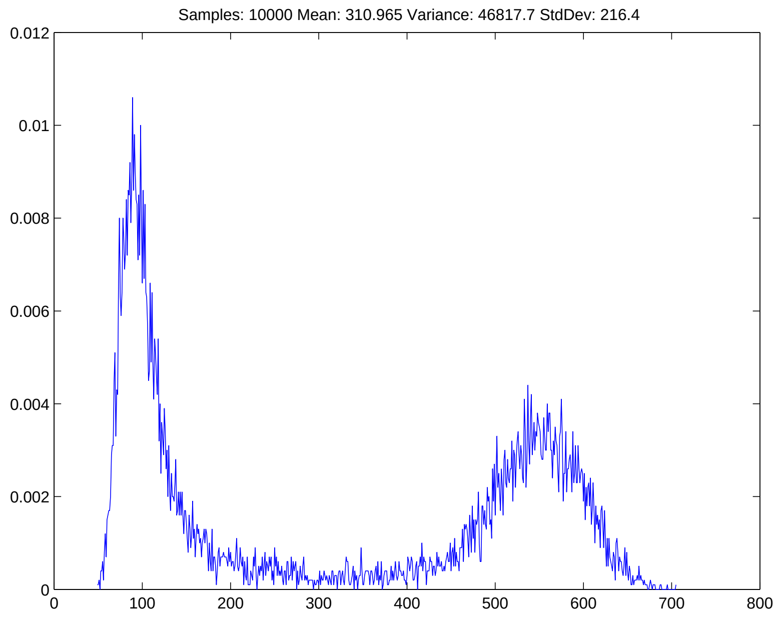

Figure 8: Histoplot_int plot for Schlögl example. This plot is based on 10, 000 samples. Note that because the values have a large range, this plot exhibits a large fluctuation.

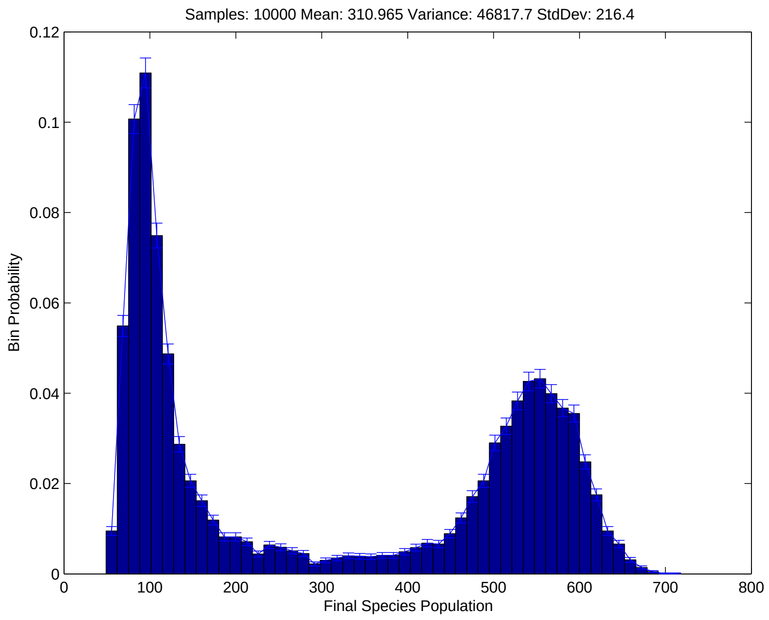

Figure 9: Histoplot_real plot for Schlögl example. This plot is based on 10, 000 samples. The default number 50 of bins is applied.

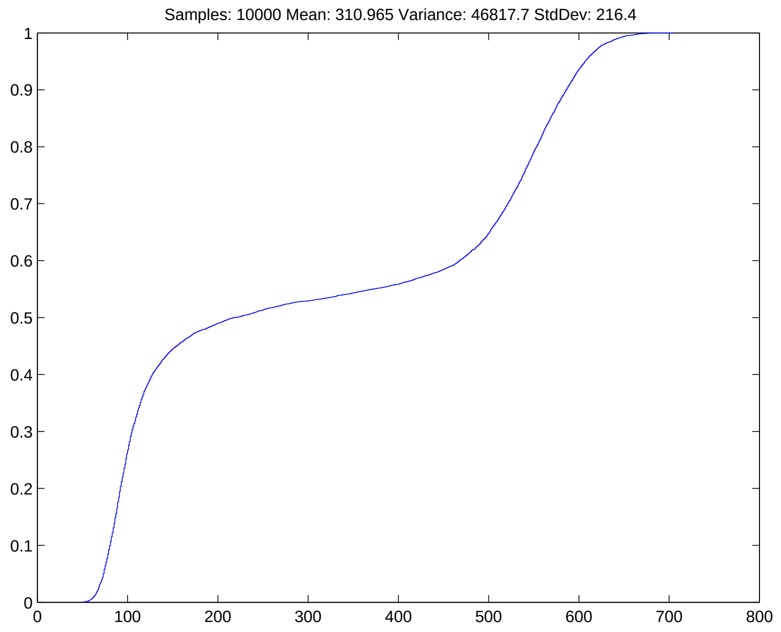

Figure 10: cdfplot plot for the Schlögl example. This plot is based on $10,000$ samples.

ables. The first two arguments are the vectors of the samples of two random variables. The numbers of samples (sizes of the two vectors) do not have to be equal. It assumes that the two vectors contain integers. If not, they will automatically be rounded to integers. The distance calculates the sum of the absolute value of the difference between the calculated probabilities of the two ensembles. Since all data are integers, the probabilities are calculated for integer values rather than bins.

histodistance_real calculates the histogram distance between two random variables. The first two arguments are the vectors of the samples of two random variables. The numbers of samples (sizes of the two vectors) do not have to be equal. The third argument is a positive integer representing the number of bins. Here we do not assume the data are integers. Thus the probabilities are calculated based on bins. The rest is the same as the function histodistance_int.

kolmogorovdistance calculates the Kolmogorov distance between two random variables. The first two arguments are the vectors of the samples of two random variables. The numbers of samples (sizes of the two vectors) do not have to be equal. The Kolmogorov distance measures the maximum distance between the measured cdfs between the two random variables.

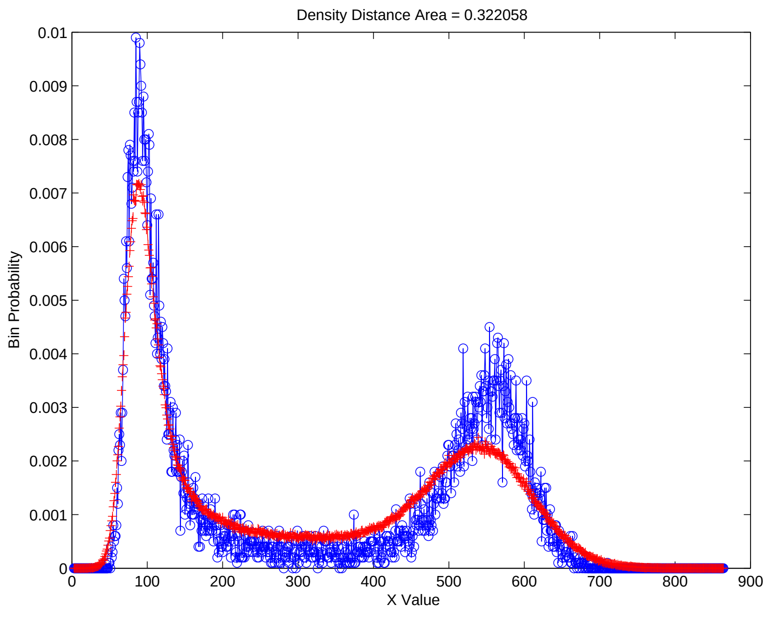

Figure 11: histodistance_int plot for the Schlögl example. This plot is based on 10,000 samples of SSA runs and explicit tau-leaping runs with tau = 0.4.

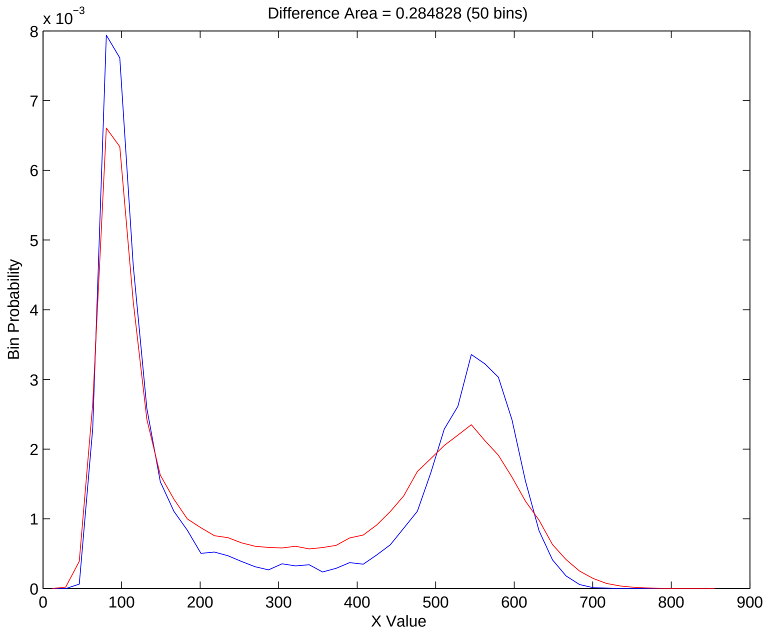

Figure 12: histodistance_real plot for the Schlögl example. This plot is based on 10,000 samples of SSA runs and explicit tau-leaping runs with tau = 0.4. The default number 50 of bins is applied.

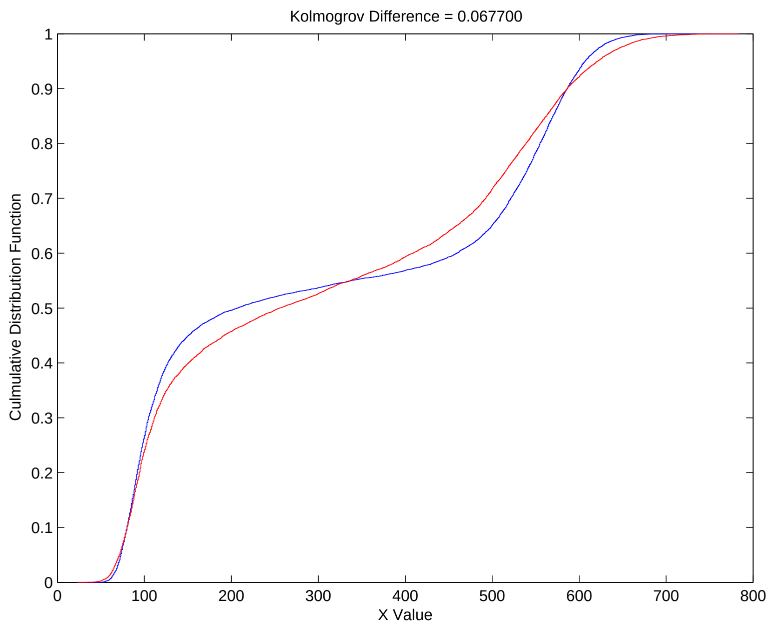

Figure 13: Kolmogorov distance plot for the Schlögl example. This plot is based on $10,000$ samples of SSA runs and explicit tau-leaping runs with tau $= 0.4$.

5.2 MPI Parallel Toolbox

Monte Carlo simulation is naturally suited to parallel computation. In StochKit we provide an MPI toolbox which enables users to parallelize the collection of Monte Carlo ensembles. Although it is part of the StochKit package, the application of this toolbox is not limited to StochKit. Any Monte Carlo ensemble collection software can be parallelized after small modifications to its code. To use this toolbox, simply follow these steps.

1. Set the environment variables to include the path for MPI.

   Example:
   setenv PATH {$PATH}:/usr/local/mpich/bin/:.

2. Compile StochKit. It is important to correctly compile StochKit for parallel computation. We recommend using SPRNG to generate the random numbers since it provides better performance and accuracy in random number generation on a parallel machine. If you choose not to use SPRNG, we made our best efforts to avoid statistical error due to the correlation between random number generation on different nodes. The results are still quite trustable. If you choose to use SPRNG, please follow the README for SPRNG (see under StochKit/Math/sprng2.0 or visit their website) to ensure that SPRNG is using its parallel version. After you

recompile SPRNG, you need to recompile StochKit. This step is important to reduce the statistical error due to the correlation between random number generation on different nodes.

3. Write the parallel code using our template. We will use the Dimerdecay model as the example to explain this step. There are two other examples: the heatshock and Schlögl models under the directory "test". [5] Suppose you have written your code for a single processor system. Copy all the files for the single processor system to a new directory. Make the following changes.

   (a) Change your main function to a normal function call.
   Example: (DimerDecay)
   Open DimerStats.cpp. Use "int DimerStats(int iterations, char outFile)" to replace the "main" fucntion line. Remove those lines concerned with the arguments and output file.*

   (b) Update the include files to reflect the above change.
   Example:
   Put "int DimerStats(int iterations, char outFile);" int DimerStats.h and include DimerStats.h in DimerStats.cpp.*

   (c) Copy parallel.cpp to the new directory and edit it as follows:

      i. Add the include file for the new problem.
      Example: Add
      #include "Random.h"
      #include "DimerStats.h"
      in parallel.cpp.

      ii. Set parameters. These parameters are the following:

         A. TotalIte: How many samples you need to simulate.

         B. NodeIte: How many samples each node will simulate at one assignment. This parameter should be chosen not be too small otherwise the message passing overhead will be large. Nor should it be too large, otherwise it may take too long for a slow node.

         C. ModelName: the directory name for the result.

      Example:
      int TotalIte=10000; // We need an ensemble of 10000 samples
      int NodeIte=500; // Each time, a node will simulate 500 samples
      char ModelName[50] = "DM"; // The name of this model

[5] Note: the directories starting with 'p' are directories for parallel computing. Otherwise, they are for single processor systems.

iii. Set the function to call. In the function "static unit_result_t do_work(unit_of_work_t work)", add your function call with work and workdir as parameters after the line *sprintf(workdir, "./result/%d%s/%d.txt", myrank, ModelName, CalledTimes);*
Example:
DimerStats(work, workdir);

4. Copy the Makefile to your new directory and edit it as follows:
   (a) Set the EXE to the name you prefer for your code.
   Example:
   EXE = p_dimer
   (b) Change parameters in the OBJS to the file name you use.
   Example:
   OBJS = parallel.o DimerStats.o ProblemDefinition.o
   (c) Change other names for your model.
5. Compile the parallel code. Type "make clean" then type "make". Now you are ready to run the parallel code.
6. To run the parallel code, use the command "mpirun -np 8 p_dimer (change it for your own model)". This means we will use 8 nodes to run the job. One will be the master node. Seven slave nodes will do the real simulation (this number has to be larger than 1). After the simulation is finished, you can find all the results in the directory "result". You can go to the subdirectory "result" to collect all the data. Just type "cat */* >result.txt". This UNIX command will save all the results in the file result.txt.

NOTE: Before and after your simulation, make sure that you have an empty directory "result". This will help to avoid mixing your simulation results.

Here are the CPU time statistics for the heatshock example using 16 nodes on the CISE/IGERT cluster in UCSB. The total number of samples is 10,000. Each subtask has 100 samples. The CPU time statistics are:

```
Node 1 working time: 65300.751900s
Node 2 working time: 71237.403293s
Node 3 working time: 71925.680539s
Node 4 working time: 72152.520430s
Node 5 working time: 72039.307552s
Node 6 working time: 64859.650003s
Node 7 working time: 72277.642849s
Node 8 working time: 71807.694310s
Node 9 working time: 72953.922793s
Node 10 working time: 73102.471573s
Node 11 working time: 62843.083589s
Node 12 working time: 73094.701832s
Node 13 working time: 72377.459407s
Node 14 working time: 72125.499277s
Node 15 working time: 64139.769542s
Total working time: 1052237.558889s
```

5.3 SBML2StochKit Converter

SBML (System Biology Makeup Language) [14] is a computer-readable format for representing models of biochemical reaction networks. SBML is applicable to metabolic networks, cell-signaling pathways, regulatory networks, and many others. Dozens of software packages support this format. Many biochemical problems are written as SBML files. For the convenience of SBML users, we provide an easy SBML to StochKit Converter using Java. Here we explain how users can translate an SBML file into the input files that StochKit needs.

1. Prepare the SBML file of your model. Our converter accepts standard Version 1 (level 1 and 2) and Version 2 SBML files with some additional requirements. These requirements are

   (a) In the tag <listOfSpecies>, give the initial amount for each species.
   Example:
   <specie name="S1" compartment="DimerDecay" initialAmount="1000" boundaryCondition="false" >

   (b) In the tag <listOfReactions>, specify the kinetic law for each reaction.
   Example:
   *<kineticLaw formula="c1*S1">*

   (c) In the tag <kineticLaw formula="c1*S1">, specify the values of the parameters.
   Example:

   <listOfParameters>
   <parameter name="c1" value="1">
   <listOfParameters>

2. Set the environment variables in your shell. For example, if you use Cshell, add the following lines to your .cshrc or run them via command line.

setenv XML_HOME <StochKit_directory>/tools/SBML2StochKit/

setenv LD_LIBRARY_PATH $XML_HOME/test: $LD_LIBRARY_PATH

3. Compile our Converter with the following command

   javac -classpath .:$XML_HOME/classes/XML.jar:$XML_HOME/classes/xerces.jar Converter.java

4. Run the Converter with the following command

   java -ms128m -mx256 -classpath .:$XML_HOME/classes/XML.jar:$XML_HOME/classes/xerces.jar Converter test.xml 10
   where "test.xml" is the file name of your SBML file and "10" is final time you want to simulate your system to.

We have provided some simple examples to demonstrate how to use the SBML2StochKit Converter. Figures 14-17 show the source files and the generated files.

References

[1] Y. Cao and L. Petzold. Trapezoidal tau-leaping formula for the stochastic simulation of chemically reacting systems. Submitted, 2005.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide.* SIAM, Philadelphia, PA, 1995.

[3] Andrew Hall. A Computational Framework for Accelerated Stochastic Simulation of Biochemical Networks. M. S. Thesis, UCSB, 2002.

[4] A. Arkin, J. Ross, and H.H. McAdams. "stochastic kinetic analysis of developmental pathway bifurcation in phage λ-infected *e. coli* cells". *Genetics*, 149:1633–1648, Aug 1998.

[5] U. M. Ascher and L. R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations.* SIAM, 1998.

[6] B.W. Brown, J. Lovato, and K. Russell. RANLIB.C *Library of C Routines for Random Number Generation.* M.D. Anderson Cancer Center, The University of Texas, 1991.

[7] C. Rao and A. Arkin. Stochastic chemical Kinetics and the quasi steady-state assumption: application to the Gillespie algorithm. *J. Chem. Phys.*, 118:4999–5010, 2003.

[8] J. Dongarra and E. Grosse. Netlib. http://www.netlib.org.

[9] M. Frankowicz, M. Moreau, P.P. Szczęsny, J. Tóth, and L. Vicente. Fast variables elimination in stochastic kinetics. *J. Phys. Chem.*, 97:1891–1895, 1993.

[10] D.T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *J. Comp. Phys.*, 22:403–434, 1976.

[11] D.T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81:2340–2361, 1977.

[12] D.T. Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. *J. Chem. Phys.*, 115(4):1716–1733, 2001.

[13] D.T. Gillespie and L.R. Petzold. Improved leap-size selection for accelerated stochastic simulation. *J. Chem. Phys.*, 119, 2003.

[14] M. Hucka, A. Finney, H.M. Sauro, H. Bolouri, J. C. Doyle, and H. Kitano. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.

[15] D. Walker J. Dongarra, R. Pozo. LAPACK++: A design overview of object-oriented extensions for high performance linear algebra. In *Proceedings of Supercomputing '93*, pages 162–171. IEEE Computer Society Press, 1993.

[16] T. Keffer and A. Vermeulen. *Math.h++ Introduction and Reference Manual.* Rogue Wave Software, Corvallis, Oregon, 1989.

[17] P. L'Ecuyer and S. Cote. Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software*, 17(1):98–111, 1991.

[18] M. Mascagni. "sprng: A scalable library for pseudorandom number generation". In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, 1999.

[19] M. Mascagni and A. Srinivasan. "sprng: A scalable library for pseudorandom number generation". *ACM Transactions on Mathematical Software*, 26:436–461, 2000.

[20] The Mathworks Inc., Natick, MA. *Using Matlab*, 1984-2002.

[21] H.H. McAdams and A. Arkin. Stochastic mechanisms in gene expression. *Proc. Natl. Acad. Sci. USA*, 94:814–819, 1997.

[22] H.H. McAdams and A. Arkin. It's a noisy business! *Trends in Genetics*, 15(2):65–69, Feb 1999.

[23] D.W. Oxtoby and N.H. Nachtrieb. *Principles of Modern Chemistry.* Saunders College Publishing, 2nd edition, 1990.

[24] R. Pozo. *Template Numerical Toolkit: An interface for scientific computing in C++.* National Institute of Standards and Technology, Gaithersburg, Maryland, 2002.

[25] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 2nd edition, 1992.

[26] M. Rathinam, Y. Cao, L.R. Petzold, and D.T. Gillespie. Stiffness in stochastic chemically reacting systems: The implicit tau-leaping method. *Journal of Chemical Physics*, 2003.

[27] J.V.W. Reynders, P.J. Hinker, J.C. Cummings, S.R. Atlas, S. Banerjee, W.F. Humphrey, S.R. Karmesin, K. Keahey, M. Srikant, and M. Tholburn. POOMA*: A Framework for Scientific Simulation on Parallel Architectures.* Los Alamos National Laboratory, 1996.

[28] J.G. Siek. *A Modern Framework for Portable High Performance Numerical Linear Algebra*, April 1999.

[29] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1997.

[30] Sun, http://java.sun.com/reference/api/index.html. *API Specifications*, 2004.

[31] T.L. Veldhuizen. *The Blitz++ User Guide.* http://www.oonumerics.org/blitz/manual/blitz.html, February 2001.

[32] Y. Cao, D. Gillespie and L. Petzold. Avoiding negative populations in explicit Poisson tau-leaping. Submitted, 2005.

[33] Y. Cao, D. Gillespie and L. Petzold. Multiscale stochastic simulation algorithm with stochastic partial equilibrium assumption for chemically reacting systems. *to appear, J. Comput. Phys.*, 2005.

[34] Y. Cao, D. Gillespie and L. Petzold. Robust and efficient stepsize control for the tau-leaping method. In preparation, 2005.

[35] Y. Cao, D. Gillespie and L. Petzold. The slow-scale stochastic simulation algorithm. *J. Chem. Phys.*, 122:014116, 2005.

A StochKit Function Reference

This section serves as a basic reference for the StochKit implementation of each of the components presented in Section 3. The general signature (including parameter and return types) for each class of functions is presented, along with a description of the purpose of each parameter and the expected semantics of functions of that class. All functions in this section are defined in the `CSE::StochRxn` namespace.

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level1" level="1" version="1">
<model name="DimerDecay">

      <listOfSpecies>
       <specie name="S1" compartment="DimerDecay" initialAmount="1000"
boundaryCondition="false" />
       <specie name="S2" compartment="DimerDecay" initialAmount="0"
boundaryCondition="false" />
       <specie name="S3" compartment="DimerDecay" initialAmount="0"
boundaryCondition="false" />
      </listOfSpecies>

      <listOfReactions>
       <reaction name="Reaction1" reversible="false">
         <listOfReactants>
           <specieReference specie="S1" />
         </listOfReactants>

         <listOfProducts>
         </listOfProducts>

         <kineticLaw formula="c1*S1">
           <listOfParameters>
             <parameter name="c1" value="1"/>
           </listOfParameters>
         </kineticLaw>
       </reaction>

       <reaction name="Reaction2" reversible="false">
         <listOfReactants>
           <specieReference specie="S1" stoichiometry="2"/>
         </listOfReactants>

         <listOfProducts>
           <specieReference specie="S2" stoichiometry="1"/>
         </listOfProducts>

         <kineticLaw formula="(c2/2.0)*S1*(S1-1)">
           <listOfParameters>
             <parameter name="c2" value="0.002"/>
           </listOfParameters>
         </kineticLaw>
       </reaction>

       <reaction name="Reaction3" reversible="false">
         <listOfReactants>
           <specieReference specie="S2" stoichiometry="1"/>
         </listOfReactants>

         <listOfProducts>
           <specieReference specie="S1" stoichiometry="2"/>
         </listOfProducts>

         <kineticLaw formula="c3*S2">
           <listOfParameters>
             <parameter name="c3" value="0.5"/>
           </listOfParameters>
         </kineticLaw>
       </reaction>
```

Figure 14: The first page of the SBML file for the DimerDecay model

Table 1: `C++` driver function reference

Name	**StochRxn**		
Signature	`SolutionHistory StochRxn(x0, t0, tf, react, opt)`		
Parameters	Name	Type	Description
	`x0`	`Vector`	Initial reactant species populations
	`t0`	`double`	Initial time
	`tf`	`double`	Final time
	`react`	`ReactionSet`	Reaction details (stoichiometric matrix, propensity function, propensity Jacobian function)
	`opt`	`SolverOptions`	Options structure produced by a call to `ConfigStochRxn`
Return Value	A `SolutionHistory` object containing a number of `SolutionPt` objects that depends on the integration time and history buffer management routine specified in `opt`.		
Purpose	Generate a single realization of the reaction system specified in `react` by propagating the system from state `x0` at time `t0` to time `tf` using the algorithms specified in `opt`.		
Name	**CollectStats**		
Signature	`EndPtStats CollectStats(runs, x0, t0, tf, react, opt)`		
Parameters	Name	Type	Description
	`runs`	`unsigned int`	Number of Realizations to perform
	`x0`	`Vector`	Initial reactant species populations
	`t0`	`double`	Initial time
	`tf`	`double`	Final time
	`react`	`ReactionSet`	Reaction details (stoichiometry matrix, propensity function, propensity Jacobian function)
	`opt`	`SolverOptions`	Options structure produced by a call to `ConfigStochRxn`
Return Value	An EndPtStats object containing a the final state vector from each realization.		
Purpose	Generate `runs` realizations of the reaction system specified in `react` by propagating the system from state `x0` at time `t0` to time `tf` using the algorithms specified in `react`. Only the final state vector from each realization is stored.		

Table 2: `C++` stepsize selection function reference

Name	**Fixed_Stepsize** **SSADirect_Stepsize** **Gillespie_Stepsize** **GillespiePetzold_Stepsize** **Cao_Stepsize**		
Signature	`double` *`Name`*`(x, a, a0, nu, tau, eps, j)`		
Parameters	Name	Type	Description
	`x`	`Vector`	Current reactant species populations
	`a`	`Vector`	Current reaction propensities
	`a0`	`double`	Sum of current reaction propensities
	`nu`	`Matrix`	Stoichiometric matrix (number of species $\times$ number of reactions)
	`tau`	`double`	Previous stepsize $(t_i - t_{i-1})$
	`eps`	`double`	Accuracy control parameter (for accelerated methods)
	`j`	`Propensity-JacobianFunc`	Pointer to function that evaluates Jacobian of reaction propensity vector
Return Value	A `double` indicating the size of the next step to take (dt). Thus, if we are executing step i at time t_i, $t_{i+1} = t_i + dt$.		
Purpose	Given information regarding the current system state and the previous stepsize, determine the size of the next step to take, using the following methods:		
	Name		Description
	`Fixed_Stepsize`		Constant stepsize equal to that specified via the `''init_step''` parameter in the `ConfigStochRxn()` call
	`SSADirect_Stepsize`		Step to the next reaction time indicated by the direct SSA method
	`Gillespie_Stepsize`		Take step according to the leap condition presented in [12]

Table 3: `C++` single stepper function reference

Name	**SSA_SingleStep** **ExplicitTau_SingleStep** **ImplicitTau_SingleStep** **ImplicitTrapezoidal_SingleStep** **TrapezoidalTau_SingleStep**		
Signature	`void` *`Name`*`(x, t, dt, a, a0, nu, pf, pjf, absTol, relTol, rxn, p)`		
Parameters	Name	Type	Description
	`x`	`Vector`	Current reactant species populations
	`t`	`double`	Current time
	`dt`	`double`	Current stepsize
	`a`	`Vector`	Current reaction propensities
	`a0`	`double`	Sum of current reaction propensities
	`nu`	`Matrix`	Stoichiometric matrix (number of species $\times$ number of reactions)
	`pf`	`Propensity-Func`	Pointer to function that evaluates reaction propensity
	`pjf`	`Propensity-JacobianFunc`	Pointer to function that evaluates Jacobian of reaction propensity vector.
	`absTol`	`double`	Absolute tolerance for implicit methods)
	`relTol`	`double`	Relative tolerance for implicit methods)
	`rxn`	`int`	reaction index of occuring reaction for SSA methods)
	`p`	`Vector`	number of reactoins for all reaction channels for tau-leaping methods)
Return Value	None. The system state vector `x` is modified in place.		
Purpose	Given the current state `x`, time `t`, and the stepsize `dt` suggested by the chosen stepsize selection function, advance `x` to time `t`+`dt`.		

Table 4: `C++` history buffer management functions

Name	**Exponential_StoreState** **NoHistory_StoreState** **OneHertz_StoreState**		
Signature	`void` *`Name`*`(hist, solPt)`		
Parameters	Name	Type	Description
	`hist`	`Solution-History`	History buffer
	`solPt`	`SolutionPt`	Object containing current time, state vector, and stepsize for possible insertion into the history buffer
Return Value	None.		
Purpose	At the discretion of the selected function, adds the current time, state, and step size to the history buffer. Actual behavior is as follows:		
	Name		Description
	`Exponential_StoreState`		Stores all solution points in the buffer, doubling the buffer's size when its capacity is exceeded
	`NoHistory_StoreState`		Stores only the final solution point in the buffer
	`OneHertz_StoreState`		Stores solution points at approximately 1 s (integration time) intervals, regardless of the stepsize choices made by the stepsize selection function. Useful for shrinking output file size when using SSA

```
        <reaction name="Reaction4" reversible="false">
          <listOfReactants>
            <specieReference specie="S2" stoichiometry="1"/>
          </listOfReactants>

          <listOfProducts>
            <specieReference specie="S3" stoichiometry="1"/>
          </listOfProducts>

          <kineticLaw formula="c4*S2">
            <listOfParameters>
              <parameter name="c4" value="0.04"/>
            </listOfParameters>
          </kineticLaw>
        </reaction>
       </listOfReactions>
</model>
</sbml>
```

Figure 15: The second page of the SBML file for the DimerDecay model

```
#include "ProblemDefinition.h"
Vector Initialize()
{
      Vector x0(3, 0.0);

      x0(0) = 1000;
      x0(1) = 0;
      x0(2) = 0;
      return x0;
}

Matrix Stoichiometry()
{
    Matrix nu(3, 4, 0.0);

   nu(0,0) = -1;    nu(1,0) = 0;    nu(2,0) = 0;

   nu(0,1) = -2;    nu(1,1) = 1;    nu(2,1) = 0;

   nu(0,2) = 2;     nu(1,2) = -1;   nu(2,2) = 0;

   nu(0,3) = 0;     nu(1,3) = -1;   nu(2,3) = 1;

   return nu;
}

Vector Propensity(const Vector& x)
{
    c1=1, c2=0.002, c3=0.5, c4=0.04;
   Vector a(4);
   a(0) = c1*x(0);
   a(1) = (c2/2.0)*x(0)*(x(0)-1);
   a(2) = c3*x(1);
   a(3) = c4*x(1);
   return a;
}

Matrix PropensityJacobian(const Vector& x)
{
    c1=1, c2=0.002, c3=0.5, c4=0.04;
   Matrix j(4,3,0.0);
   j(0,3) = c1;
   j(1,3) = (c2/2.0)*(x(0)-1);
   j(2,3) = c3;
   j(3,3) = c4;
   return j;
}
```

Figure 16: The problem definition file generated by the SBML2StochKit Converter: `ProblemDefinition.C++`.

```
#include "StochRxn.h"
#include "ProblemDefinition.h"
#include "Vector.h"
#include "Matrix.h"
#include <stdlib.h>

using namespace CSE::Math;
using namespace CSE::StochRxn;
Vector Propensity(const Vector& x);
Matrix PropensityJacobian(const Vector& x);

int main(int argc, const char* argv[])
{
   try{
      int numRuns;
      std::string outFile;
      double TimeFinal = 100

      if (argc == 3) {
         numRuns = atoi(argv[1]);
         outFile = argv[2];
      }
      else {
         std::cerr << "Usage:  dimerstats <# runs> <output file>";
         exit(EXIT_FAILURE);
      }

      // Set up the problem:
      Vector X0 = InitialState()
      Matrix nu = Stoichiometry()
      ReactionSet rxns(nu, Propensity, PropensityJacobian);

      // Configure solver
      SolverOptions opt;
      opt.stepsize_selector_func = SSADirect_Stepsize;
      opt.single_step_func = SSA_SingleStep;
      opt.progress_interval = 1000000;
      opt.initial_stepsize = 0.001;
      opt.absolute_tol = 1e-6;
      opt.relative_tol = 1e-5;

      //Make the Run and report results
      EndPtStats stats = CollectStats(numRuns, x0, 0, TimeFinal, rxns,
opt)
      WriteStatFile(stats, outFile);
      std::cerr << "Done.\n";
   }

   catch (const std::exception& ex) {
      std::cerr << "\nCaught " << ex.what() << '\n';
   }

   return 0;
}
```

Figure 17: The simulation code generated by SBML2StochKit Converter: `Dimer.C++`.