SOFTWARE FOR MULTI-LEVEL MONTE-CARLO SIMULATION OF STOCHASTIC BIOCHEMICAL KINETICS

by

Dexter Barrows

April 28, 2014

A thesis

presented to the Department of Mathematics

in partial fulfillment of the

requirements for the degree of

Bachelor of Science

in the Program of

Mathematics and its Applications

at Ryerson University

Supervisor: Dr. Silvana Ilie

Abstract

Stochastic models for systems of biochemical reactions are essential to the field of Systems Biology. While the Chemical Master Equation provides accurate predictions of future states of well-stirred biochemical systems, the solutions to this model are too analytically complex to obtain for realistic systems, and direct numerical methods are likewise too computationally complex to be a feasible solution. Thus, Monte Carlo-type methods such as the Stochastic Simulation Algorithm (SSA) are utilized by scientists to obtain results consistent with solutions to the Chemical Master Equation but with drastically reduced complexity. This thesis discusses our program for the simulation of stochastic models of well-stirred biochemical kinetics, the Modelling Arrays of Reaction Software (MARS). Among other simulation methods, we implemented a very recent and advanced numerical strategy to estimate the average behaviour of the biochemical system, the multilevel Monte Carlo (MLMC) tau-leaping method. We compared the predictions of the MLMC with those of the exact SSA. The results showed excellent agreement, with the multilevel Monte Carlo tau-leaping method demonstrating a greatly reduced running time.

Acknowledgements

To

my Mom

without whom

nothing I achieve

would be possible

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Chemical reactions form the basis for virtually all biological processes, and understanding them is crucial to the progress of scientific discovery. Single chemical reactions have long been studied, quantified, qualified, experimented upon, and been thoroughly understood by the scientific community. However, it is not a single chemical reaction that can serve as the basis for something as complex as a biological organism, but rather dynamic systems of reactions with overlapping species that cannot be unlinked. The chemical systems associated with biological processes are referred to as systems of biochemical reactions.

Mathematical models of biochemical reaction systems provide valuable analytical tools that can be used to understand and predict the behaviour of such systems. These models aim to determine the evolution of dynamic populations of species over elapsing time. The building blocks of all such models are differential equations, which relate functions to their derivatives. They are used to construct two overarching types of models: ones which are continuous in time, and ones that take discrete values depending on time [14]. The classical approach to modelling chemical reactions uses the former approach, employing systems of differential equations. These models are also deterministic, meaning they do not take into account the "noise", or elements of randomness, present in actual systems of biochemical reactions. The prototypical model of this type is the Reaction Rate Equation (RRE) [5]. It has since been accepted that systems that do take noise into account are necessary in order to provide realistic results [10],[14]. The stochastic models may be continuous or discrete.

Fundamentally, all stochastic models work on the same idea: they do not provide exact

deterministic behaviour of a system, but rather a probabilistic outcome. The finest model of well-stirred biochemical kinetics is the Chemical Master Equation (CME) [11],[5], a discrete and stochastic model. The solution to the CME, really a system of coupled ordinary differential equations (ODEs), is a probability distribution of potential systems states at a given future time. While a solution to the CME is the ideal way to represent systems of well-stirred biochemical reactions, this is not analytically or numerically possible for all but the simplest of systems – the CME is too complex for these approaches. However, single, exact realizations of the CME can be numerically generated by the Stochastic Simulation Algorithm (SSA), due to Gillespie [4],[5], which uses discrete species populations to probabilistically simulate a series of reactions at variable-size time steps. To reduce the computational complexity, these exact trajectories can be in turn approximated through the consolidation of multiple reactions in fixed-size intervals, known as the tau-leaping method, introduced by Gillespie in [7]. A further reduction is possible when large populations are present. Then, the molecular numbers may be represented as continuous populations. The reduced stochastic continuous model is known as Chemical Langevin Equation (CLE) [7]. Good approximations of the probability distribution provided by the solution to the CME can be obtained by generating a large number of independent trajectories using SSA, the tau-leaping, or the pathwise simulation of CLE, usually on the order of 10,000 trajectories. Often, the mean and standard deviation of the results constitute the quantities of interest for applications.

This work will discuss the theoretical framework required for stochastic modelling of well-stirred systems of biochemical reactions, the derivation of the CME, SSA, tau-leaping, CLE, and RRE models, and present the conditions under which these modelling approaches and simulation methods are valid in Chapter 2. Chapter 3 will introduce an advanced method for generating linked trajectories that greatly reduce the computational complexity of the algorithm for the same accuracy, the multilevel Monte Carlo (MLMC) tau-Leaping [1],[2].

Chapter 4 will describe and provide documentation for our program which implements all of the aforementioned methods, the Modelling Arrays of Reactions Software (MARS). MARS allows the user, through the simulation method of their choosing, to obtain or plot predictive data for the evolution in time of the species populations in a biochemical system contained in a Systems Biology Markup Language (SBML) file. It has two primary function modes: it can generate a single trajectory to display the general system behaviour, or can use on the order of 10,000 independent trajectories to predict

the mean behaviour of the system. Additionally, Chapter 4 outlines the many optional user-specified options that MARS can accept, which contributes to making the implementation quite robust. Appendix A contains all original source code, as well as a link to a version of the code with the additional open-source libraries required for operation [9],[3].

Lastly, Chapter 5 will present numerical results obtained using MARS to simulate several systems of biochemical reactions. The SSA, tau-leaping, the numerical methods for the CLE, and RRE will be compared through simulation of the classic Michaelis-Menten system [7], and the Schlögl model [8]. The MLMC method will be compared to SSA through the simulation of the Goldbeter-Koshland Switch, a cyclical reaction system, and a potassium channel model [12].

Chapter 2

Background

2.1 Motivation for Stochastic Modelling of Biochemical Systems

Systems Biology at its core is the study of the complex interactions between components of organisms. Its goal is to describe the function and behaviour of all parts of a biological system, and in turn the organism as a whole. It is a broad field with many areas of study. One such area focuses on modelling interactions of populations of biochemical molecules. The field has evolved over the years to move beyond the study of single biomolecules into the interaction of many different types of these molecules engaged in reactions, the aim of which is to describe and eventually predict the system's behaviour.

Originally, these dynamic systems have been modelled as sets of coupled ordinary differential equations (ODEs). This model is called the Reaction-Rate Equation (RRE). On a larger scale, this approach works quite well, but it is intrinsically ill-suited to describing systems at the cellular level where the number of molecules may be much, much smaller. As this model is continuous and deterministic, it is unable to take into account the "noise" in a system. In fact, it has been well-established that biochemical kinetics at a cellular level contain a natural degree of unpredictability or randomness (stochasticity), that makes accurate description of their behaviour simply beyond the grasp of the RRE. As such, it has become generally accepted that stochastic models are required in order to present a realistic mathematical model of cellular-level biochemical kinetics.

However, problems arise. Chief among them is that these stochastic models are computationally very demanding. In order to reduce complexity, a key assumption must be

made: cells (containers) must be treated as well-mixed, eliminating the concern over the spacial component of the system. This is not out of the ordinary, deterministic models make this assumption as well. Further, this approach gives accurate results while allowing the state of a system at a given point in time to be represented as a vector containing the number of each type of molecule present. Even with this assumption, only simple networks can be feasibly approached with standard tools. In this case, it is possible to generate a probability distribution of the state of the system over time. Models of more complexity, even ones as simple as consisting of more than single-molecule reactions, quickly become analytically intractable. It is important to note that any systems that would be of practical interest belong to this latter class. Then software tools are needed to numerically approximate the solution of the mathematical models of biological processes. Mathematically, the state of a system is modelled as a Markov process.

2.2 Modelling and Simulation of Biochemical Systems

2.2.1 Stochastic Discrete Model

Model: Chemical Master Equation

With the need for stochastic modelling established, we must then turn to the mechanics of constructing such a model. First, the mathematical representation of a biochemical systems must be formalized. As per the previous section, consider a thermally and spatially isolated container containing biochemical molecules which we will be treating as well-stirred. Let there be N such species $\{S_1, \ldots, S_N\}$ that interact through a system of M chemical reactions $\{R_1, \ldots, R_M\}$. The state of the system can then be represented such that the number of each individual type of molecule S_i at a given time t is denoted $X_i(t)$, with the so-called state vector of the system being represented as $\mathbf{X}(t) \equiv (X_1(t), \ldots, X_N(t))$. We also assume that the system begins in initial state $\mathbf{X}(t_0) = \mathbf{x}_0$ at initial time t_0.

Of course, this representation contains only population data. How, then, can it be sufficient in representing a system that also includes many other types data, including seemingly important information about the speed and position of individual particles? The answer comes from particle theory and the assumption that we are dealing with a well-stirred system. They allow us to assume that the system, being well-stirred, consists of collisions where the overwhelmingly majority are of an elastic, non-reactive nature. Thus, the positions of the individual particles are randomized across the entirety of

the container, and the velocities of these particles are randomized consistent with the Maxwell-Boltzmann distribution. We can then ignore these reactions, instead focusing on the ones that will alter our definition of the system: the populations of each type of molecule. This dramatically simplifies the problem.

The focus of our efforts then turns to examining the effect each of the aforementioned M reactions has on the system. These are less simply represented, requiring a twofold approach. First, each reaction will have a state-change vector in the form $\mathbf{v}_j \equiv (v_{1j}, \ldots, v_{Nj})$, where each v_{ij} represents a change in the population of species S_i from the occurrence of reaction R_j. Because each v_{ij} represents the effect a particular reaction will have on each molecule type in the system as a whole, a reaction R_j occurring will have the effect that a system will instantaneously move from state $\mathbf{x}$ to state $\mathbf{x} + \mathbf{v}_j$.

A simple example: take a system consisting of three species A, B, and C. The initial amounts of these molecules are known, and can be represented as a state vector written as

$$\mathbf{X}(t) = \begin{bmatrix} X_1(t) \\ X_2(t) \\ X_3(t) \end{bmatrix}.$$

We define $X_1(t)$ to be the number of molecules of A at time t, $X_2(t)$ to be the number of molecules of B at time t, and $X_3(t)$ to be the number of molecules of C at time t. Now take the reaction

$$A + B \rightarrow C,$$
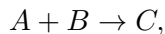
representing molecules A and B combining to form molecule C. We can construct a state change vector $\mathbf{v}$ for this reaction in the form

$$\mathbf{v} = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}.$$

This shows that every time the reaction occurs, one molecule each of species A and B are consumed, and one molecule of C is formed. Hence, after this reaction fires, the system would be in the state

$$\mathbf{X}(t + dt) = \mathbf{X}(t) + \mathbf{v} = \begin{bmatrix} X_1(t) - 1 \\ X_2(t) - 1 \\ X_3(t) + 1 \end{bmatrix},$$

representing the next state of the system after a time dt has elapsed, which would be the time between when the system entered the initial state $\mathbf{X}(t)$ and when the reaction fires. Here we are working with the assumption that the time it takes for the reaction itself to occur is infinitesimally small compared to dt, that is the reactions are instantaneous events.

The question that now naturally arises is how large or small should dt be? For this, we turn to the second quantity used to represent a reaction R_j: the propensity function. A propensity function serves a very intuitive purpose; it takes into account the amount of each species present as a reactant in a particular reaction, along with that reaction's experimentally-derived reaction rate constant, to help predict how much time will elapse before that reaction is likely to fire. It does not predict the amount of time itself, instead providing an idea of how likely that reaction is to occur relative to other reactions in the system.

Formally, a propensity function, denoted $a_j(X(t))$, has the property that the probability of reaction j occurring on the time interval $[t, t + dt)$ is $a_j(X(t))dt$. The general form of a reaction's propensity function is the same, but will differ based on the type of the reaction.

First Order Reactions of the form

$$S_i \xrightarrow{c_j} Product(s),$$

have a propensity function directly proportional to the reaction constant c_j, treated here as a scaling factor, multiplied by the amount of the species, $X_i(t)$. The associated propensity function is

$$a_j(X(t)) = c_j X_i(t).$$

Second Order Reactions of the form

$$S_i + S_k \xrightarrow{c_j} Product(s), \text{ with } i \neq k,$$

have a propensity function such that it is directly proportional to the reaction constant c_j, treated here as a scaling factor, multiplied by the number of ways a single collision can occur given the amounts of each species. From combinatorics, we know this to simply be these amounts multiplied by each other, in this case $X_i(t)X_k(t)$. Thus, the associated propensity function is

$$a_j(X(t)) = c_j X_i(t) X_k(t).$$

Dimerisation   Reactions of the form
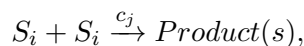
$$S_i + S_i \xrightarrow{c_j} Product(s),$$

have a propensity function such that it is directly proportional to the reaction constant c_j, treated here as a scaling factor, multiplied by the number of ways a single collision can occur given the amounts of the species. Again from combinatorics, we know this to be equivalent to choosing two identical objects from a common pool of size n: $\binom{n}{2} = \frac{1}{2}n(n-1)$, in this case $\frac{1}{2}X_i(t)(X_i(t) - 1)$. The associated propensity function will then be

$$a_j(X(t)) = c_j \frac{1}{2} X_i(t)(X_i(t) - 1).$$

In the rare case of higher order (polymerisation) reactions of this type, we again turn to combinatorics to obtain the general equation for choosing r identical objects from a common pool of size n: $\binom{n}{r} = \frac{1}{r!}X_i(t)(X_i(t) - 1)(X_i(t) - 2)\ldots(X_i(t) - (r-1))$. The associated propensity function here would then be

$$a_j(X(t)) = c_j \frac{1}{r!} X_i(t)(X_i(t) - 1)(X_i(t) - 2)\ldots(X_i(t) - (r-1)).$$

Now that we have the tools necessary to completely describe both components of a biological system, species and reactions, we can start to examine their behaviour.

We start by introducing a new definition: $P(\mathbf{x}, t)$ is the probability that the state vector $X(t)$ is in the particular state $\mathbf{x}$ at time t. Now, assuming that we know the probability that the system is in a particular state $\mathbf{x}$ at time t, we attempt to determine the next state the system will enter in to at time $t + dt$. In this way we will establish a recurrence relation between states at adjacent time intervals. Notice then that there are only two ways that a system can be in state $\mathbf{x}$ at time $t + dt$; either it was already in that state at time t and no reaction took place over the interval $[t, t + dt)$, or it was in state $\mathbf{x} - \mathbf{v}_j$ and

reaction j occurred during $[t, t + dt)$. We eliminate the possibility of multiple reaction occurring during $[t, t + dt)$ by imposing the restriction that dt is so small that only a single reaction can occur.

With that in place, we turn momentarily to probability theory. Consider the probability of an event A occurring preceded by one of the events $B_0, B_1, \ldots, B_{M+1}$. Further, suppose that the B_j set of events have two properties: that they are exhaustive (one of them *must* occur) and disjoint (no more than one may occur). In other words, exactly one of the events in the set will occur. Then the *law of total probability* states that

$$P(A) = \sum_{j=0}^{M+1} P(A|B_j)P(B_j). \tag{2.1}$$

This means that the probability of A occurring is the sum of each probability that A will occur given that B_j has already occurred multiplied by the probability that B_j occurs. When applied to the problem at hand, we have event A being the event that the system is in state $\mathbf{x}$ at the time $t + dt$ (the later event), and the set of B_j events being the various events that can occur during $[t, t + dt)$ to get the system into state $\mathbf{x}$ (the earlier set of events). More precisely, define B_0 as the event that no reaction occurs, B_j for $1 \leq j \leq M$ being the event of reaction R_j occurring (recall that there are M reactions in a system), and B_{M+1} being the event that the system is originally in a state for which the current state $\mathbf{x}$ cannot be reached with the occurrence of a single reaction. Of course this last event is an impossibility given that we have restricted dt to prevent having to consider precisely this occurrence, a fact that will be accounted for in short order.

In the meantime, we can observe that we have already part of equation (2.1) defined in terms of our system. The component $P(A|B_j)$ being the probability that the system is in state $\mathbf{x}$ at time $t + dt$ given that it was in state $\mathbf{x} - \mathbf{v}_j$ at time t is just the probability that reaction R_j has occurred during $[t, t + dt)$, which is exactly the propensity function for that reaction with $\mathbf{x} - \mathbf{v}_j$ as its argument. Hence we obtain

$$P(A|B_j) = a_j(\mathbf{x} - \mathbf{v}_j)dt. \tag{2.2}$$

With B_j for $1 \leq j \leq M$ taken care of, we turn to B_0 and B_{M+1}. Since $P(A|B_0)$ is just the probability that no reaction occurs, and the sum of probabilities in a probability space must sum to 1, we derive that

$$P(A|B_0) = 1 - \sum_{j=1}^{M} a_j(\mathbf{x})dt. \tag{2.3}$$

Since B_{M+1} cannot occur, we can get

$$P(A|B_{M+1}) = 0 \tag{2.4}$$

We now have all the components we need in order to fill in equation (2.1) to represent our system. Substituting equations (2.2), (2.3), and (2.4) into (2.1), we obtain

$$P(\mathbf{x}, t + dt) = \left(1 - \sum_{j=1}^{M} a_j(\mathbf{x})dt\right) P(\mathbf{x}, t) + \sum_{j=0}^{M} a_j(\mathbf{x} - \mathbf{v}_j)dt P(\mathbf{x} - \mathbf{v}_j, t).$$

Simply rearrange it to give

$$\frac{P(\mathbf{x}, t + dt) - P(\mathbf{x}, t)}{dt} = \sum_{j=1}^{M} [a_j(\mathbf{x} - \mathbf{v}_j)P(\mathbf{x} - \mathbf{v}_j, t) - a_j(\mathbf{x})P(\mathbf{x}, t)].$$

Now if we take the limit as $dt \to 0$, it is clear that the right hand side of the equation is the derivative of $P(\mathbf{x}, t)$. We have derived the Chemical Master Equation (CME) [11]

$$\frac{dP(\mathbf{x}, t)}{dt} = \sum_{j=1}^{M} [a_j(\mathbf{x} - \mathbf{v}_j)P(\mathbf{x} - \mathbf{v}_j, t) - a_j(\mathbf{x})P(\mathbf{x}, t)]. \tag{2.5}$$

It is important to note that this is actually a system of coupled ODEs with the discrete state vector $X(t)$ having a great many possible values, each with its own ODE. It quickly becomes clear that it will only be possible to analytically examine or even numerically solve such a system of ODEs for very simple systems; ones of any complexity are again simply infeasible to attempt.

Stochastic Simulation Algorithm: Exact Simulation of the Chemical Master Equation

As discussed, the CME is simply too complex to be approached analytically or computationally for anything other than extremely simple systems. How, then, can we approach the CME in order to derive any information of importance for more complex (realistic) systems? The approach to take is thus: since the probability space of the potential states of the state vector is too complex to compute in its entirety, we simulate a *single* exact

realization of the state vector, a trajectory. This can be done as many times as needed to generate a statistically significant result. While the multi-trajectory generation can be very computationally intensive, as will be discussed later, it will still be much less demanding than solving the CME directly. This approach, called the Stochastic Simulation Algorithm (SSA, also know as Gillespie's algorithm after its inventor, Daniel T. Gillespie) enables us to produce results of value with a much lower resource requirement [4].

In order to derive the algorithm, some new notation is required. We will let $P_0(\tau|\mathbf{x}, t)$ be the probability that no reaction takes place during the interval $[t, t+\tau)$, given an initial known system state $X(t) = \mathbf{x}$. Now we will consider the interval $[t, t+\tau+dt)$. This interval can be split at $t+\tau$ into two separate intervals $[t, t+\tau)$ and $[t+\tau, t+\tau+dt)$, which are independent of each other. From probability theory we know that if there are two independent events A and B, then $P(A \cap B) = P(A)P(B)$. Using this, the new notation, and equation (2.3), we can then write

$$P_0(\tau + d\tau|\mathbf{x}, t) = P_0(\tau|\mathbf{x}, t)P_0(d\tau|\mathbf{x}, t+\tau).$$

We then can apply equation (2.3) to yeild

$$P_0(\tau + d\tau|\mathbf{x}, t) = P_0(\tau|\mathbf{x}, t)\left(1 - \sum_{j=1}^{M} a_j(\mathbf{x})d\tau\right).$$

Taking a similar approach to the one taken while deriving the CME, the second form above can be rearranged to give

$$\frac{P_0(\tau + d\tau|\mathbf{x}, t) - P_0(\tau|\mathbf{x}, t)}{d\tau} = -P_0(\tau|\mathbf{x}, t)\sum_{j=1}^{M} a_j(\mathbf{x}).$$

Now taking the limit as $d\tau \to 0$ leads to

$$\frac{dP_0(\tau|\mathbf{x}, t)}{dt} = -P_0(\tau|\mathbf{x}, t)\sum_{j=1}^{M} a_j(\mathbf{x}). \tag{2.6}$$

This is a first-order ODE. As the probability of no reaction taking place over a period of time 0 is necessarily 1, then we can assert that

$$P_0(0|\mathbf{x}, t) = 1. \tag{2.7}$$

Solving equation (2.6) with initial condition (2.7) will yield

$$P_0(\tau|\mathbf{x}, t) = e^{-\sum\limits_{j=1}^{M} a_j(\mathbf{x})\tau} \tag{2.8}$$

This intermediate equation will be used shortly. For now, we will define another new quantity. Let $P(\tau, j|\mathbf{x}, j)d\tau$ be the probability that the next reaction to occur in the system will have reaction index j, where again $1 \leq j \leq M$, and that it will occur during the interval $[t + \tau, t + \tau + d\tau)$. If we treat these events as A and B respectively, and notice that the probability of a reaction taking place during $[t + \tau, t + \tau + d\tau)$ will be the same as the probability of *no* reaction taking place over $[t, t + \tau)$, then again from probability theory we can use $P(A \cap B) = P(A)P(B)$, along with out first definition and the definition of the propensity function to write

$$P(\tau, j|\mathbf{x}, j)d\tau = a_j(\mathbf{x})d\tau P_0(\tau|\mathbf{x}, t).$$

Note that $P(\tau, j|\mathbf{x}, j)$ represents a probability density function over *two* random variables; the reaction index of the next reaction, and the time that will elapse until that reaction occurs. This means that we need a way to unlink these variables else we run into the same type of problem that exists with the CME: the complexity of the resulting equation will simply be too high to approach either analytically or numerically.

But, by canceling out the $d\tau$ factors on each side of the equation and, and making use of equation (2.8), this expression can be rewritten as

$$P(\tau, j|\mathbf{x}, j) = a_j(\mathbf{x})e^{-\sum\limits_{j=1}^{M} a_j(\mathbf{x})\tau},$$

then again to give

$$P(\tau, j|\mathbf{x}, j) = \left(\frac{a_j(\mathbf{x})}{\sum\limits_{j=1}^{M} a_j(\mathbf{x})}\right)\left(\sum\limits_{j=1}^{M} a_j(\mathbf{x})e^{-\sum\limits_{j=1}^{M} a_j(\mathbf{x})\tau}\right). \tag{2.9}$$

Now notice that (2.9) has been written such that the leftmost factor of the right hand side of the equation will be the density function for the reaction index j, while the rest of that side represents an exponential distribution of the type that frequently represents elapsed time between events. As such, we have successfully uncoupled our two random variables. Specifically, each of these quantities can now be chosen from separately gen-

erated random variables chosen over a uniform $(0, 1)$ sample.

Thus the following algorithm, the Stochastic Simulation Algorithm [4], can now be implemented.

Stochastic Simulation Algorithm Initialize the system state by taking $X(t_0) = \mathbf{x}_0$. Given the system state $X(t) = \mathbf{x}$, the next state $X(t + \tau)$ and next time $t + \tau$ will be determined by:

1. Evaluate each individual $a_k(\mathbf{x})$ for $1 \leq k \leq M$ and the sum $a_{sum} := \sum_{j=1}^{M} a_j(\mathbf{x})$.

2. Retrieve two random numbers r_1 and r_2 from a uniform $(0, 1)$ distribution.

3. Determine the reaction index j to use by finding the smallest value satisfying the equation $\sum_{k=1}^{j} a_k(\mathbf{x}) > r_1 a_{sum}(\mathbf{x})$.

4. Set time step size $\tau = \ln(1/r_2)/a_{sum}(\mathbf{x})$.

5. Set the next system state $X(t + \tau) = \mathbf{x} + \mathbf{v}_j$ and update system time t to $t + \tau$.

Repeat 1 to 5 until the simulation time t exceeds the desired simulation time, then exit.

The termination condition used could of course be something else entirely, however the one here is the one implemented in the software described in a later chapter. It is important to note, again, that this algorithm is exact in the sense that it realizes one possible trajectory of the system exactly (it is error-free). For this reason, it is the primary algorithm that is implemented in the aforementioned software.

Tau-leaping: Approximate Simulation of the Chemical Master Equation

Typically, the SSA is not efficient in practice. While computationally feasible to perform, it is still relatively expensive. Many studies focused on the development of techniques that allow this method to be sped up in exchange for minor reductions in accuracy. One such technique, know as tau-leaping, will be discussed here.

A major source of computational complexity found in SSA is the fact that the propensity functions (propensities) for each reaction, $a_j(X(t))$ for $1 \leq j \leq M$, must be recomputed at each new time step t. However, if the amounts of each species that contribute to any given propensity function are relatively large compared to the the change in those

populations from step-to-step, then the relative value of the propensities will not change very much. In fact, as we are only dealing with lower-order reactions, the typical change in any given population step-over-step will be only be a few molecules. This means that if we were to allow several, but not too many, reactions to occur during a given time interval, then we can make the assumption that the propensities will remain virtually unchanged, allowing in some cases drastic increases in simulation speed. There will, in reality, be small changes in the values of the propensities, which is where error is introduced. This upper-bound limitation on the value of tau is know as the Leap Condition. There are many methods of choosing a tau that is large enough to have an impact on the speed of the simulation, but not exceed the Leap Condition. One such method will be discussed later in this paper. For now, we will work on the assumption that the value of tau being used is a good value.

We can now develop a formula that makes use of the assumption. We pick a fixed time interval τ and count how many reactions will occur. From probability theory, we know that if we wish to know how many times an event A will occur over a given time period τ, this can be determined by drawing a Poisson random variable from a distribution with Adt events expected to occur over the infinitesimal time period dt. Let $\mathcal{P}(A, \tau)$ represent such a variable.

Now to tie this to the problem at hand. From the definition of a propensity function, $a_j(X(t))dt$ is the probability that reaction R_j should occur over the infinitesimal interval $[t, t + dt)$. Hence, the number of times reaction R_j should be expected to occur over a given interval $[t, t+\tau)$ with a Poisson distribution will then be $\mathcal{P}(a_j(X(t)), \tau)$. We then assume that each reaction R_j *will* take place that many times over this interval, so then by drawing M such independent random Poisson variables, one for each reaction, and each with the first parameter corresponding to that reaction's propensity function, we obtain

$$X(t + \tau) \doteq X(t) + \sum_{j=1}^{M} \mathbf{v}_j \mathcal{P}_j(a_j(X(t)), \tau). \tag{2.10}$$

By taking this approach, the overall speed of the simulation can increase greatly. However, as mentioned, there are reductions in accuracy. It is unlike SSA in that it is not an exact realization of a single trajectory of the CME, but rather an accurate and often efficient approximation of one. An additional problem also arises: taking a too large value for the step tau can lead to negative population values (obviously problematic). It

is important to reiterate: tau must be chosen carefully.

Then, the algorithm to implement this explicit tau-leaping procedure can be written as follows [7].

Tau-leaping Algorithm   Initialize the system state by taking $X(t_0) = \mathbf{x}_0$. Given the system state $X(t) = \mathbf{x}$, the next state $X(t + \tau)$ and next time $t + \tau$ will be determined by:

1. Select a value for τ based on the system state $\mathbf{x}$ that satisfies the Leap Condition.

2. For each j in $1 \leq j \leq M$, draw a Poisson random variable p_j according to $p_j = \mathcal{P}(a_j(\mathbf{x}), \tau)$.

3. Update the system's state vector $X(t)$ to $X(t + \tau) = \mathbf{x} + \sum_{j=1}^{M} p_j \mathbf{v}_j$ and system time t to $t + \tau$.

Repeat 1 to 3 until the simulation time t exceeds the desired simulation time, then exit.

Note that in the above procedure, it is advisable to recompute tau at every step. If a single value for tau is chosen at the beginning of the simulation, and were held constant throughout, several serious problems may arise:

- The chosen value for tau might satisfy the Leap condition at the beginning of the simulation, but could exceed this bound with changes to the system state over the course of the simulation.

- The value could cause a leap that is too large, which may lead to negative populations.

- The leap could be large relative to a potential future state of the system as to cause the error inherent to the method to exceed desired limitations.

As such, using a fixed value for tau should be regarded with caution.

2.2.2 Stochastic Continuous Models

Model: Chemical Langevin Equation

While the Tau-leaping method of approximating an exact trajectory of the SSA has the benefit of much faster simulation speed, further approximation can allow even greater

gains. Generation of Poisson random variables can be expensive, and so a large contributor to simulation speed. However, we know from statistical theory that a Poisson random variable with a particular mean and variance can be well approximated by a normal random variable with the same mean and variance as long as they are much greater than one. Let normal random variable with mean μ and variance σ^2 be denoted by $\mathcal{N}(\mu, \sigma^2)$. For a Poisson random variable $\mathcal{P}(A, \tau)$, with mean and variance $A\tau$, this would mean as long as $A\tau \gg 1$, we can approximate $\mathcal{P}(A, \tau) \simeq \mathcal{N}(A\tau, A\tau)$.

Recall the particular Poisson random variables in the tau-leaping method are of the form $\mathcal{P}(a_j(X(t)), \tau)$, one for each of the M reactions. The mean and variance for such a variable would be $a_j(X(t))\tau$. Using the above, if each $a_j(X(t))\tau \gg 1$ for any $1 \leq j \leq M$, then each Poisson random variable will be well approximated by a normal random variable such that

$$\mathcal{P}(a_j(X(t)), \tau) \simeq \mathcal{N}(a_j(X(t))\tau, a_j(X(t))\tau). \tag{2.11}$$

We can then substitute (2.11) into (2.10) to obtain

$$X(t + \tau) \doteq X(t) + \sum_{j=1}^{M} \mathbf{v}_j \mathcal{N}_j(a_j(X(t))\tau, a_j(X(t))\tau). \tag{2.12}$$

And using the fact that $\mathcal{N}(\mu, \sigma^2) = \mu + \sigma\mathcal{N}(0, 1)$, this becomes

$$X(t + \tau) \doteq X(t) + \sum_{j=1}^{M} \mathbf{v}_j \left(a_j(X(t))\tau + \sqrt{a_j(X(t))}\sqrt{\tau}\mathcal{N}_j(0, 1) \right). \tag{2.13}$$

Now, (2.13) is the Euler-Maruyama numerical method applied to a Stochastic Differential Equation (SDE). The SDE is derived by what is known as a multidimensional Wiener process [6]. We will take a brief departure from the core problem at hand to examine define a Wiener process, to examine how to approach them computationally, and how it relates to equation (2.12).

A Wiener process is a time-dependant random variable over a time interval defined as follows.

Definitions   Let $W(t)$ be a random variable and $[0, T]$ be a time interval such that $t \in [0, T]$. $W(t)$ is a *Wiener process* if

- $W(0) = 0$ with probability 1.

- Given times a and b such that $0 \leq a < b \leq T$, the random variable represented by the incremental change $W(b) - W(a)$ is normally distributed with a mean 0 and variance of $b - a$.

- Given additional times c and d such that $0 \leq a < b < c < d \leq T$, the increments of $W(b) - W(a)$ and $W(d) - W(c)$ are independent of each other.

It is important to note here that the increment in the second condition $W(b) - W(a)$ can be equivalently expressed as

$$W(b) - W(a) \simeq \sqrt{b-a}\mathcal{N}(0,1). \tag{2.14}$$

Being that we are dealing here with discrete-time stepping in our simulations, we could use (2.14) to numerically generate a Wiener trajectory.

First we can divide the $[0, T]$ interval into N subintervals, where N is a positive integer, by defining the size of each subinterval as $\tau = T/N$. Each $W(t_j)$ is the state of the Wiener process at time t_j (and so at each $j\tau$ division of the time interval), and can be represented using (2.14) as

$$W(t_j) - W(t_{j-1}) = \sqrt{\tau}\mathcal{N}(0,1), \quad \text{for } j = 1, 2, \ldots, N. \tag{2.15}$$

Bringing this all into context of a normal random variable approximation of tau-leaping, we note that we can substitute (2.15) into (2.13) and get

$$X(t+\tau) \doteq X(t) + \sum_{j=1}^{M} \mathbf{v}_j a_j(X(t))\tau + \sum_{j=1}^{M} \mathbf{v}_j (W_j(t+\tau) - W_j(t)). \tag{2.16}$$

Now if we, as before, rearrange the equation and take the limit as $\tau \to 0$, we derive

$$dX(t) \doteq \sum_{j=1}^{M} \mathbf{v}_j a_j(X(t))dt + \sum_{j=1}^{M} \mathbf{v}_j \sqrt{a_j(X(t))}dW_j(t). \tag{2.17}$$

This SDE is known as the Chemical Langevin Equation(CLE) [7].

As noted while transitioning from tau-leaping to the CLE, we have implicitly begun to approximate using *continuous* populations. This happened when we approximated each reaction's Poisson random variable as a normal random variable - a type which is not discrete. We did this under a particular assumption, that the populations were all large enough that each of their propensities had the property $a_j(X(t))\tau \gg 1$. We

have not placed any restrictions on the population such that their propensities will have this property, so let us instead provide a guideline: in practice each of the propensities $a_j(X(t))$ will be satisfactorily large as to satisfy this property if each population $X_i(t)$ is of size $X_i(t) \geq 100$.

Applying Euler-Maruyama to the Chemical Langevin Equation

Now we discuss how to numerically approximate the solution of the CLE. One such numerical technique was already touched on during the derivation of the CLE. This approach is the Euler-Maruyama method for the numerical solution of a SDE, such as the CLE. The continuous time interval $[0, T]$ is divided into N subintervals by selecting the time steps t_i, with $0 = t_0 < t_1 < \ldots < t_N \leq T$. Then we apply (2.16) at the grid points to give

$$X(t_{n+1}) \doteq X(t_n) + \sum_{j=1}^{M} \mathbf{v}_j a_j(X(t_n))\tau + \sum_{j=1}^{M} \mathbf{v}_j \sqrt{a_j(X(t_n))}(W_j(t_{n+1}) - W_j(t_n)),$$

where $\tau = t_{n+1} - t_n$. Further apply (2.15) to give

$$X(t + \tau) \doteq X(t) + \sum_{j=1}^{M} \mathbf{v}_j a_j(X(t))\tau + \sum_{j=1}^{M} \mathbf{v}_j \sqrt{a_j(X(t))\tau}\mathcal{N}_j(0, 1). \tag{2.18}$$

This equation is known as the Langevin Leaping formula. Note that the Langevin Leaping formula and the CLE are mathematically equivalent, but now we can apply the same algorithm for determining successive systems states that we used to implement Tau-leaping. However, there is another condition that must be met. Recall that each of the propensities must be large enough that $a_j(X(t))\tau \gg 1$. Then the chosen value for τ must satisfy two conditions:

- Assumption 1: The value of τ must be small enough so that each of the propensities will not change by a large amount (the Leap Condition).

- Assumption 2: The value of τ and the species populations must be large enough so that each of the propensities will have the property $a_j(X(t))\tau \gg 1$.

Clearly, these two assumptions are at odds. However, since Assumption 2 relies on both τ and the species populations, we can usually ensure that a value for τ exists that satisfies both assumptions if each $X_i(t) \geq 100$.

Under the conditions above, the Chemical Langevin Equation model is valid and we can implement the Langevin Leaping formula as the Langevin Leaping Algorithm (LLA).

Langevin Leaping Algorithm   Initialize the system state by taking $X(t_0) = \mathbf{x}_0$. Given the system state $X(t) = \mathbf{x}$, the next state $X(t + \tau)$ and next time $t + \tau$ will be determined by:

1. Select a value for τ based on the system state $\mathbf{x}$ that satisfies Assumptions 1 and 2, and ensure that population sizes are at least 100.

2. For each j in $1 \leq j \leq M$, draw a normal random variable $\mathcal{N}_j$ according to $\mathcal{N}_j = \mathcal{N}(0, 1)$.

3. Update the system's state vector $X(t)$ to $X(t + \tau) = \mathbf{x} + \sum_{j=1}^{M} \mathbf{v}_j a_j(\mathbf{x})\tau + \sum_{j=1}^{M} \mathbf{v}_j \sqrt{a_j(\mathbf{x})\tau}\mathcal{N}_j$ and time t to $t + \tau$.

Repeat 1 to 3 until the simulation time t exceeds the desired simulation time, then exit.

The advantage of this method is increased speed from a combination of many reactions being "leapt" over during each step, and reduced computational complexity from needing to generate only normal random variables instead of less easily generated Poisson random variables.

2.2.3   Deterministic Continuous Models

Model: Reaction Rate Equation

In classical chemistry, chemical kinetics are modelled similarly to what has been discussed in previous sections. There is, however, a key difference: while mathematical models of biochemical kinetics typically account for "noise" in a system, the classical model does not. It instead offers a continuous and deterministic system of ODEs know as the Reaction-Rate Equation (RRE).

Note that the CLE is a system of SDEs consisting of two components: a deterministic component depending solely on the reactions' propensity functions, and a stochastic component.

Recall the equation for the CLE:

$$dX(t) \doteq \sum_{j=1}^{M} \mathbf{v}_j a_j(X(t))dt + \sum_{j=1}^{M} \mathbf{v}_j \sqrt{a_j(X(t))}dW_j(t).$$

The deterministic component $f(X(t)) = \sum_{j=1}^{M} v_j a_j(X(t))$ is known as the drift coefficient, while the stochastic term $g_j = v_j \sqrt{a_j(X(t))}$ is known as the diffusion coefficient.

The CLE may be reduced under the assumption that the system is approaching the thermodynamic limit. The thermodynamic limit is defined as the limit taken as as each species X_i, collectively $X(t)$, and the system volume Ω all approach infinity such that individual species concentrations X_i/Ω all stay approximately constant. As the system grows, so do the propensity functions, and will do so in direct proportion to the size of the system. The effect this growth has on the CLE is the left side of the equation and drift coefficients growing at an equivalent rate, while the diffusion coefficient will grow at the square root of this rate. Consequentially, the size of this last diffusion term will quickly become negligible in relation to the other two terms.

Under the thermodynamic limit, the CLE will reduce to the RRE by discarding the now negligibly small diffusion term and rearranging to give:

$$\frac{dX(t)}{dt} = \sum_{j=1}^{M} \mathbf{v}_j a_j(X(t)). \tag{2.19}$$

This is, as mentioned, a system of continuous and deterministic ODEs. As such, it can be solved by typical ODE solvers.

In practice, the assumption we make to apply the thermodynamic limit and reduce the CLE to the RRE is valid for cases in which each $X_i \geq 1000$.

Stiff Ordinary Differential Equation Solvers   *Stiffness* of a system refers to the heterogeneity of time scales present over the progression of time. A stiff system will have at least two (usually widely) varying time-scales, the fastest of which exhibits stability [13]. In the context of biochemical reactions, stiffness must be carefully considered. It is very common for systems of biochemical reactions to contain reactions of varying speeds, resulting in a system of ODEs evolving on time scales (varying). Ignoring this can result in information about the slow reactions becoming buried under the information from faster reactions, leading to inaccurate simulations. In practice, systems of biochemical

reactions are typically stiff.

Mathworks' MATLAB software contains ODE solvers that are specifically designed to handle stiff systems of ODEs. The program discussed later can make use of a stiff solver, specifically the "ode15s" solver.

Non-stiff Ordinary Differential Equation Solvers   Non-stiff ODE solvers can still solve stiff systems. However, they take many steps to do what a stiff ODE solver can do in a lot fewer steps, so they are very inefficient when applied to such systems. By default, the program discussed later will use the standard "ode45" solver to plot a system. But since we are considering most systems of biochemical reactions to be stiff, the option to use the "ode15s" solver in its stead should be considered carefully by the user in cases where the program run time is large.

Chapter 3

Numerical Methods

3.1 Multilevel Monte Carlo Tau-Leaping

Consider the problem of estimating the average behaviour of a system of biochemical reactions within a certain error. This can be straightforwardly implemented by generating 10,000 trajectories of the system simultaneously using SSA, tau-leaping, LLA, or similar methods, then averaging their results. From statistics, we know the the error is bounded by the inverse square root of the number of trajectories. In the case of generating 10,000 trajectories, this amounts to an error of about 10^{-2}. This method, while easy to implement and conceptually sound, can also be very computationally demanding, especially if using SSA to generate the trajectories, and in particular for stiff systems.

A recent development in the modelling of biochemical kinetics, the multilevel Monte Carlo (MLMC) tau-leaping, due to Anderson and Higham [1], uses coupled trajectories to dramatically reduce the computational complexity of multi-trajectory simulation for the same accuracy requirement as the original tau-leaping method. This method requires only a small trade-off in accuracy when compared to large numbers of SSA trajectories, and allows the maximum error to be specified. Instead of using brute force to generate many trajectories with the aim of determining the average behaviour, the MLMC instead estimates the mean behaviours of the system by using carefully chosen trajectories of different levels of coarseness, and coupling them together. The reduction in computational complexity is twofold: carefully choosing the step size of the trajectories allows for fewer trajectories to be generated, and linking pairs of trajectories allows for reuse of random variable samples drawn, so that fewer are required.

Let $\mathbb{E}f(X(T))$ be the expected value of the quantity of interest for a system of bio-chemical reactions integrated over the interval $[0,T]$. Here f is a polynomial and X is the state vector of the biochemical system. Further, let ε be the desired error tolerance, and M be an integer of $O(1)$ (typically chosen between 2 and 5). Define $h_l = TM^{-l}$, where $l \in \{0,1,...,L\}$ to be the step size for the grid on level l, and define Z_l as the approximate stochastic process generated on the grid of step size h_l. Here, L is chosen by taking $L = O(|\ln(\varepsilon^{-1})|)$, so that the finest step h_L will be of order $O(\varepsilon)$. Further, the difference between the exact expected value $\mathbb{E}f(X(T))$ and the estimator of trajectories taken with this finest step size $\mathbb{E}f(Z_L(T))$ will then also be of $O(\varepsilon)$.

The task is to estimate $\mathbb{E}f(Z_L(T))$. Following [1], we note that:

$$\mathbb{E}f(Z_L(T)) = \mathbb{E}[f(Z_0(T))] + \sum_{l=1}^{L} \mathbb{E}[f(Z_l(T)) - f(Z_{l-1}(T))] \tag{3.1}$$

Now, we define $\widehat{Q}_0$ as the estimator for $\mathbb{E}[f(Z_0(T))]$ using n_0 generated trajectories, and $\widehat{Q}_l$ as the estimator for $\mathbb{E}[f(Z_l(T)) - f(Z_{l-1}(T))]$ using n_l generated trajectories. These are calculated as follows:

$$\widehat{Q}_0 = \frac{1}{n_0} \sum_{i=1}^{n_0} f(Z_{l_0,[i]}(T)), \tag{3.2}$$

and

$$\widehat{Q}_l = \frac{1}{n_l} \sum_{i=1}^{n_l} (f(Z_{l,[i]}(T)) - f(Z_{l-1,[i]}(T))). \tag{3.3}$$

We denote $Z_{l,[i]}$ the i-th realization on a grid of step size h_l. The simulation of the trajectories needed to estimate (3.2) is straightforward: simply generate n_0 trajectories with step size h_0 and take their average. The simulation of the trajectories needed for (3.3) requires the generation of coupled trajectories, with the goal of reducing the variance. It requires the processes Z_l and Z_{l-1} to be computed in such a way that they are coupled. Let $\mathcal{P}_j$ be the Poisson processes generated by reaction j. The Poisson process $\mathcal{P}_j$ may be obtained as the sum of two independent Poisson processes, $\mathcal{P}_{j,1}$ and $\mathcal{P}_{j,2}$, where $\mathcal{P}_{j,1}$ is the first such process and $\mathcal{P}_{j,2}$ would be the second. A coupled process with $\mathcal{P}_j$ may be obtained by the summation of the Poisson process $\mathcal{P}_{j,1}$ and a new, independant Poisson process $\mathcal{P}_{j,3}$. Further let $\eta(s) = \lfloor s/h_l \rfloor h_l$, so it is a step function increasing by h_l as time increases over the interval $[0,t]$. Finally, define $a \wedge b = min\{a,b\}$. Then, the processes are linked using the following equations:

$$Z_l(t) = Z_l(0) + \sum_{j=1}^{R} \mathcal{P}_{j,1}\left(\int_0^t a_j(Z_l(\eta_l(s))) \wedge a_j(Z_{l-1}(\eta_{l-1}(s)))ds\right)\mathbf{v}_j$$

$$+ \sum_{j=1}^{R} \mathcal{P}_{j,2}\left(\int_0^t a_j(Z_l(\eta_l(s))) - a_j(Z_l(\eta_l(s))) \wedge a_j(Z_{l-1}(\eta_{l-1}(s)))ds\right)\mathbf{v}_j$$

$$Z_{l-1}(t) = Z_{l-1}(0) + \sum_{j=1}^{R} \mathcal{P}_{j,1}\left(\int_0^t a_j(Z_l(\eta_l(s))) \wedge a_j(Z_{l-1}(\eta_{l-1}(s)))ds\right)\mathbf{v}_j$$

$$+ \sum_{j=1}^{R} \mathcal{P}_{j,3}\left(\int_0^t a_j(Z_{l-1}(\eta_{l-1}(s))) - a_j(Z_l(\eta_l(s))) \wedge a_j(Z_{l-1}(\eta_{l-1}(s)))ds\right)\mathbf{v}_j$$

$$(3.4)$$

Note that the summation of the independent Poisson processes $\mathcal{P}_{j,1}$ with rate r_1 and $\mathcal{P}_{j,2}$ with rate r_2 gives a Poisson process with a rate equal to $r_1 + r_2$. Thus in equation (3.4) we obtain Poisson processes with the rate given by the tau-leaping method. Here, and for the remainder of this chapter, let R be the number of reactions so as not to confuse it with the parameter M. It is important to note that as $\mathcal{P}_{j,1}$ is the same in each equation, it can be calculated once and reused. This is why the Z_l and Z_{l-1} processes are now coupled, and why the computational complexity of the problem is reduced.

Assuming two level step sizes h_l and h_{l-1} have been calculated, the coupled trajectories are computed by first setting $Z_l(0) = X(0)$, $Z_{l-1}(0) = X(0)$, $t = 0$. Then following Anderson and Higham [1], we derive the algorithmic form (3.4) as:

1. For $k = 0, ..., M - 1$:

    (a) Set:

    - $A_{j,1} = a_j(Z_l(t + k \cdot h_l)) \wedge a_j(Z_{l-1}(t))$
    - $A_{j,2} = a_j(Z_l(t + k \cdot h_l)) - A_{j,1}$
    - $A_{j,3} = a_j(Z_{l-1}(t)) - A_{j,1}$

    (b) For each reaction j where $1 \leq j \leq R$, set:

    - $\Lambda_{j,1} = \mathcal{P}(A_{j,1} \cdot h_l)$
    - $\Lambda_{j,2} = \mathcal{P}(A_{j,2} \cdot h_l)$
    - $\Lambda_{j,3} = \mathcal{P}(A_{j,3} \cdot h_l)$

    (c) For each reaction j where $1 \leq j \leq R$, set:

- $Z_l(t + (j + 1) \cdot h_l) = Z_l(t + j \cdot h_l) + \sum_{j=1}^{R}(\Lambda_{k,1} + \Lambda_{k,2})\mathbf{v}_j$
- $Z_{l-1}(t + (j + 1) \cdot h_l) = Z_{l-1}(t + j \cdot h_l) + \sum_{j=1}^{R}(\Lambda_{k,1} + \Lambda_{k,3})\mathbf{v}_j$

2. Update the system's time t to $t = t + h_{l-1}$.

Repeat 1 to 2 until the system time t exceeds the desired simulation time, then exit.

This is only one linked pair of trajectories, more are needed to satisfy the error criteria. In order to determine the required number of trajectories satisfying the required accuracy, the system requires scaling. This is a process by which all quantities in the system (species populations and reaction rates) must be normalized, to within an order of magnitude, against the largest quantity in the system. This quantity, denoted by V, is taken to be the largest initial species population. While the species populations are expected to vary over the course of simulation, the initial populations serve as a best guess given that future states of the system are unknown. All species populations are scaled by a parameter α_i such that $O(1) = V^{-\alpha}X_i(0)$ for each species S_i. This is equivalent to evaluating $\alpha_i = \log_V X_i(t)$. Similarly for the reaction rates c_j, each is scaled according to $c_j = O(1)V^{\beta_j}$, equivalent to evaluating $\beta_i = \log_V c_j$. The scaling factor for the reaction as a whole, γ is calculated by taking

$$\gamma = \max_{i,j:\, \mathbf{v}_{ij} \neq 0}\{\beta_j + v_j \cdot \alpha - \alpha_i\}, \tag{3.5}$$

where $\mathbf{v}_{ij}$ is the entry for species S_i in the state change vector for reaction R_j, and v_j is a vector of the number of each species consumed by reaction R_j. Another important parameter, ρ, is similarly calculated by taking

$$\rho = \min_{i,j:\, \mathbf{v}_{ij} \neq 0}\{\alpha_i\}, \tag{3.6}$$

Then, the number of single trajectories needed for the estimation of (3.2), n_0, is determined by taking

$$n_0 = V^{-\rho}V^{\gamma}\epsilon^{-2} \tag{3.7}$$

and the number of coupled trajectories necessary to estimate (3.3) with accuracy ε, n_l, is obtained by taking

$$n_l = V^{-\rho}V^{\gamma}(L - l_0)h_l\epsilon^{-2}. \tag{3.8}$$

Now, $\mathbb{E}f(Z_L(T))$ can be estimated to within an error of $O(\epsilon)$ by approximating the

individual expected values in the right hand side of (3.1). It should be noted that the implementation of MLMC can be done with any desired number of levels, with more levels providing more accuracy but being more computationally complex. It is then up to the implementer to determine the optimal number of levels. Three levels were recommended in the the source material, so that is how many are used in the later described software.

Computational results for MLMC are contained in the Numerical Results chapter.

Chapter 4

MARS Software

4.1 The Aim of MARS

The goal of Modelling Arrays of Reactions Software (MARS) is to enable modelling of systems of biochemical reactions using any of the methods outlined in the previous chapter easily. MARS will accept a Systems Biology Markup Language (SBML) file containing a system of biochemical reaction in a single container. This format was chosen due primarily to it becoming a standard in the Systems Biology field. Additionally, it is currently being developed and supported, and good libraries are available to make interacting with SBML files simple. More information on SBML can be found at `http://sbml.org/`.

MARS will enable user interaction with an SBML file as specified above in two primary ways: it can produce a species versus time graph to chart species populations over reactions' progression, or produce a histogram of the resulting populations after 10,000 separate trajectories. With either method, any number of species in the system can be graphed, allowing the user to plot only those species that are of interest, and results can be graphed on the same set of axes or on a separate set of axes for each species. The default algorithm used is SSA. The user has the option to specify Tau-leaping, LLA (denoted as CLE in the software), RRE, or MLMC to model populations over time, or specify Tau-leaping or LLA to generate the histogram.

4.2   Implementation and Capabilities

In order to ensure that MARS will run on as many platforms as possible without the need for porting, it is written in MATLAB code. It includes the low-level libraries necessary to interact with SBML files on both Microsoft Windows and Apple OS X (Intel x86-based) platforms, providing a complete and integrated package on those platforms. MARS is also capable of running on Linux systems, however the user must build and install the free open-source libSBML library available at `http://sbml.org/Software/libSBML` [3] before running MARS. MATLAB 2013b or later and the MATLAB Parallel Computing Toolbox are also required. Additionally, MARS makes use of the SBMLToolbox [9], already included in the source code.

MARS runs as a single function, with the arguments provided to it dictating its behaviour. By default, MARS is set to simply run a single trajectory using SSA and return the numerical data for time step sizes and populations at those steps. It will use a default simulation time of 50 and record the system's state every 20 steps. Note that the latter setting will have no effect on the RRE method as it provides a continuous instead of discrete graph, or on the MLMC method as it produces results at specified static intervals. The user is able to specify:

- Operation mode

  - Populations vs. time graph(s)
  - Multi-trajectory histogram(s) and average behaviour plot(s)

- The simulation time (how long the simulation should run before terminating in terms of the system's internal time)

- How often to record system state (after how many steps)

- Intermediate results are output for diagnostic purposes (verbose mode)

- Whether all species' data will be graphed on the same set of axes or if each species will be graphed on a separate set of axes

- Which modelling and simulation method to use

  - Populations vs. time graph(s): SSA (default), Tau-leaping, LLA (CLE), RRE
  - Multi-trajectory histogram(s) and average system behaviour plot(s): SSA (default), Tau-leaping, LLA (CLE), or MLMC (average behaviour only)

- To use the system's GPU for multi-trajectory generation on histogram (runs in parallel)

- To simply return data without producing a graph (default), or graph the results in accordance with the above options for separate/combined graphs and system state recording step size

As can be seen, MARS is very flexible and can accommodate most use-cases. More capabilities and refinements are in the pipeline, and are outlined in a later section.

4.3 Usage

MARS accepts a combination of single-name and name-value pair arguments, most of which are optional. The most basic use-case, obtaining graphing data for a system of chemical reactions in the SBML file 'system.xml' with a simulation time of 50 and a system state recording step size of 20 steps, is undertaken by simply entering

```
>> [Y,t] = MARS('system.xml');
```

into MATLAB. Note, the ';' at the end of the command is to suppress the output of first returned argument to the terminal. Here `Y` is the name of the matrix to contain the species amount data, and `t` is the vector containing the time-step data. The matrix `Y` will be formatted such that each row contains the species amount data for a particular species after each step (so the first row will contain the population amount of species 1 after step 1, them step 2, and so on) and each row will correspond to that numbered species. The time step vector will contain the system's elapsed time up to the end of that step. Note that the MLMC method will also produce data in this format, but the populations values are averages over many trajectories instead of single data points.

To run 10,000 trajectories and obtain the resulting data (ideal for a histogram plot), again using the default simulation time of 50:

```
>> [Y,Means,Std_dev] = MARS('system.xml','Hist');
```

Where `Y` is the name of the matrix to contain the species final amounts data. It will be formatted such that each column contains that numbered species' amount data after each completed trajectory, where each entry in the column shows the number of that species present after that numbered trajectory has run its course (so the first column will

contain the number final population value of species 1 after trajectory 1, then trajectory 2, and so on). Additionally, `Mean` and `Std_dev` are the names of matrices to contain the mean behaviour and standard deviation data at each step (default is 100 steps). They will be in the same format as the output from the single trajectory generation data, that is formatted such that each row contains the species amount data for a particular species after each step (so the first row will contain the population amount of species 1 after step 1, them step 2, and so on) and each row will correspond to that numbered species.

MARS accepts arguments after the initial SBML file name argument. This is done as follows:

```
>> [Y,t] = MARS('system.xml',options);
```

where `options` is the list of arguments you want MARS to accept. The lists of valid singular and name-value pair arguments options are in the following sections.

4.3.1 Optional Single Arguments

Singular arguments are ones that do not have any required or optional context, but rather enable a single capability on their own. They are as follows:

'Verbose'	To run the program in verbose mode, enabling diagnostic outputs while setting up and running the simulation.
'Split'	To split the results graph into individual graphs for each species present.
'RRE'	To use the RRE method.
'Stiff'	To use a stiff ODE solver (specifically "ode15s") when using the RRE method. Note that this argument will have no effect if RRE is not being used.
'GPU'	To use the system's CUDA-enabled GPU (to generate or plot histogram data). Note that the 'GPU' argument automatically implies the 'Hist' argument, so you need not also include it.

4.3.2 Optional Name-value Pair Arguments

MARS implements the name-value pair paradigm found in many MATLAB programs for several argument types. These are as follows:

`'Time', tfinal`	To run the simulation for a specific period of time, where `tfinal` is an integer representing the length of the time desired.
`'Record', steps`	To record the system state (which will also affect how fine-grained the graph will appear) every certain number of steps, where `steps` is an integer representing how many steps you want to allow the algorithm to take before next recording the state of the system. Note that this argument is incompatible with the `'Hist'` argument, so `'Record'` will be ignored if `'Hist'` is also used.
`'Tau', value`	To use the Tau-leaping method with the value for τ being `value`. If `value` is not specified, a static value is estimated.
`'CLE', value`	To use the LLA (CLE) method with the value for τ being `value`. If `value` is not specified, a static value is estimated.
`'Steps', value`	To record system state at regularly spaced intervals with `value` being the number of such intervals. If `value` is not specified, a default of 100 steps will be used. This option is only relevant to parallel methods not running on a GPU, so multi-trajectory generation using SSA, Tau-leaping, or CLE methods running on parallel CPUs or CPU cores, and the MLMC method, will all be affected.
`'MLMC', value`	To use the MLMC method with the value for M being `value`. If `value` is not specified, the default value of $M = 3$ is used.
`'Error', value`	To use a specific tolerance with the MLMC method with the value for ε being `value`. If `value` is not specified, the default value of $\varepsilon = 10^{-2}$ is used.
`'Graph', species`	To produce a graph of all species data in the system, either with single-trajectory generation or histogram generation. To plot the data only for specific species in the system, follow the `'Graph'` argument with a vector - `species` - containing the index numbers of the species you want to graph using the order of species from the SBML file provided.

Any valid combinations of arguments are also allowed. Order is not important except for the name-value pair sets of arguments.

Chapter 5

Numerical results

5.1 Michaelis Menten Model

The Michaelis Menten model [7] consists of four species engaged in one reversible reaction (which is split into two reactions), and one non-reversible reaction. This model has been very well studied, so its expected behaviour is known, making it a good model to use for testing software implementations.

The relevant model data, the reactions, their propensities, and the corresponding reaction rates, are in Table 5.1.

	Reactions	Propensities	Reaction rates
R_1	$S + E \xrightarrow{c_1} S_E$	$a_1(\mathbf{x}) = c_1 SE$	$c_1 = 0.00166$
R_2	$S_E \xrightarrow{c_2} S + E$	$a_2(\mathbf{x}) = c_2 S_E$	$c_2 = 0.0001$
R_3	$S_E \xrightarrow{c_3} P + E$	$a_3(\mathbf{x}) = c_3 S_E$	$c_3 = 0.1$

Table 5.1: Michaelis Menten model

The initial population values for $(S, E, S_E, P)^T$ are $(301, 120, 0, 0)^T$. Single trajectories were generated over an interval of $[0, 30]$ using SSA, tau-leaping, LLA/CLE, and RRE methods using the MARS software.

The results are presented in Figure 5.1.
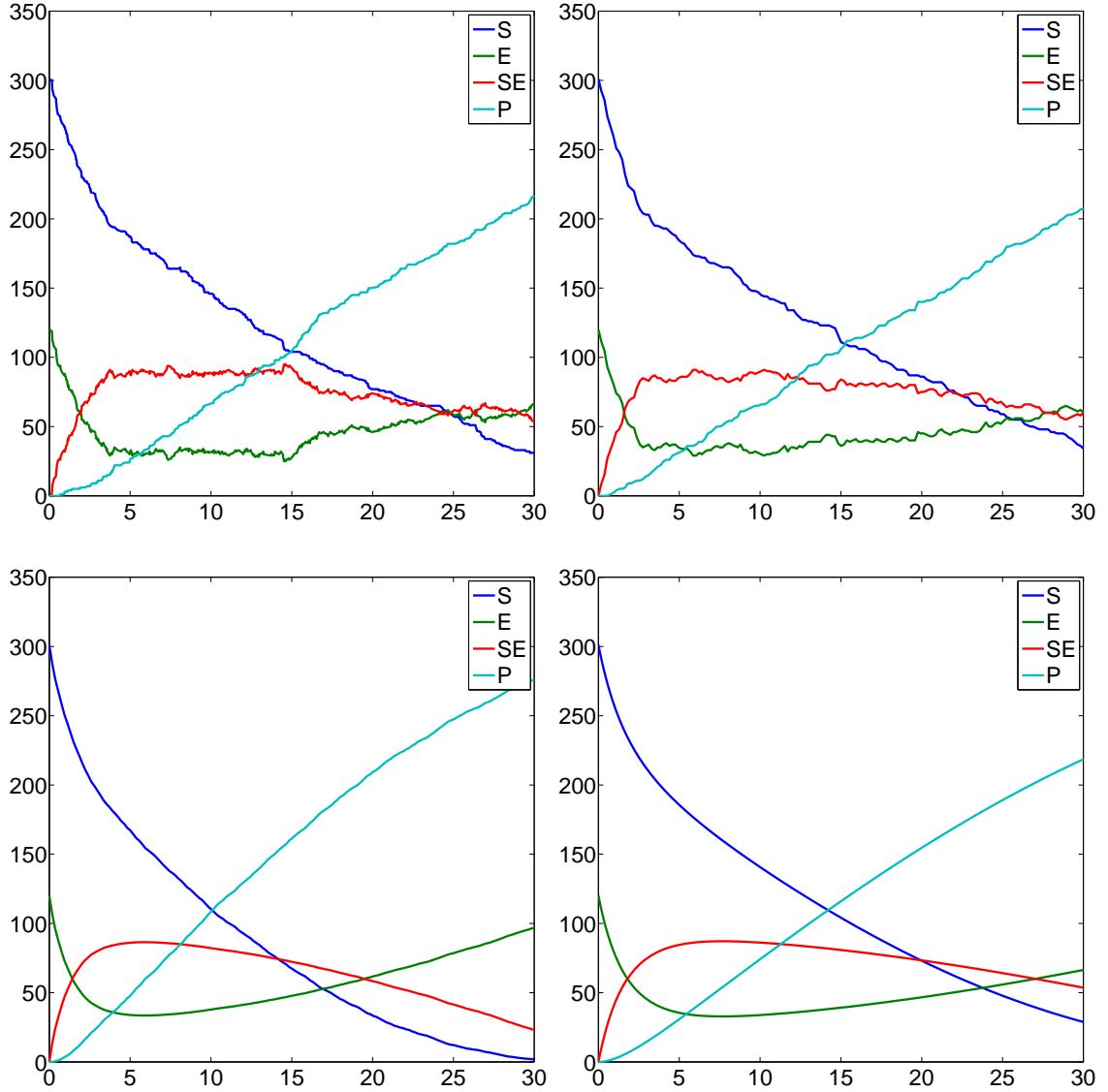
Figure 5.1: Integration of the Michaelis Menten model over the interval $[0, 30]$ using SSA (top left), tau-leaping (top right), LLA/CLE (bottom left), and RRE (bottom right). Vertical axis shows population amounts, horizontal axis shows simulation time. The value of τ employed by tau-leaping and LLA/CLE was automatically determined by MARS.

Using the same initial population values of $(301, 120, 0, 0)^T$, histograms of the end population were then produced by generating results from 10,000 trajectories. The same methods were applied, with the exception of GPU generation of trajectories being sub-

stituted for RRE, which would be ineligible for this test as it gives a deterministic prediction.

The results are in the following Figure 5.2.

Figure 5.2: Histogram of population amounts computed on 10,000 trajectories, at the final time $T = 30$, after simulation of the Michaelis Menten model over $[0, 30]$ using SSA (top left), tau-leaping (top right), LLA/CLE (bottom left), and GPU (bottom right). The vertical axis shows frequency, the horizontal axis shows final population amounts. The value of τ utilized by tau-leaping and LLA/CLE was automatically determined by MARS.

5.2 Schlögl Model

The Schlögl model [8] consists of 3 species engaged in two reversible reactions (which are split into four reactions). This model contains a bifurcation based on the value of the initial population of one of the species. This will be discussed in the next section.

The relevant model data presented in Table 5.1.

	Reactions	Propensities	Reaction rates
R_1	$A + 2X \overset{c_1}{\to} 3X$	$a_1(\mathbf{x}) = c_1 A X (X-1)/2$	$c_1 = 3 \times 10^{-3}$
R_2	$3X \overset{c_2}{\to} A + 2X$	$a_2(\mathbf{x}) = c_2 X(X-1)(X-2)/6$	$c_2 = 10^{-4}$
R_3	$B \overset{c_3}{\to} X$	$a_3(\mathbf{x}) = c_3 B$	$c_3 = 10^{-3}$
R_4	$X \overset{c_4}{\to} B$	$a_3(\mathbf{x}) = c_4 X$	$c_4 = 3.5$

Table 5.2: Schlögl model

Additionally, the populations of species A and B are held constant, making X the only species of interest. The model is integrated with initial population values for $(A, B, X)^T$ of $(10^5, 2 \times 10^5, 248)^T$. Single trajectories were generated over the interval $[0, 15]$ employing SSA, tau-leaping, LLA/CLE, and RRE methods of the MARS software.

Numerical results for the Schlögl model are shown in Figure 5.3.
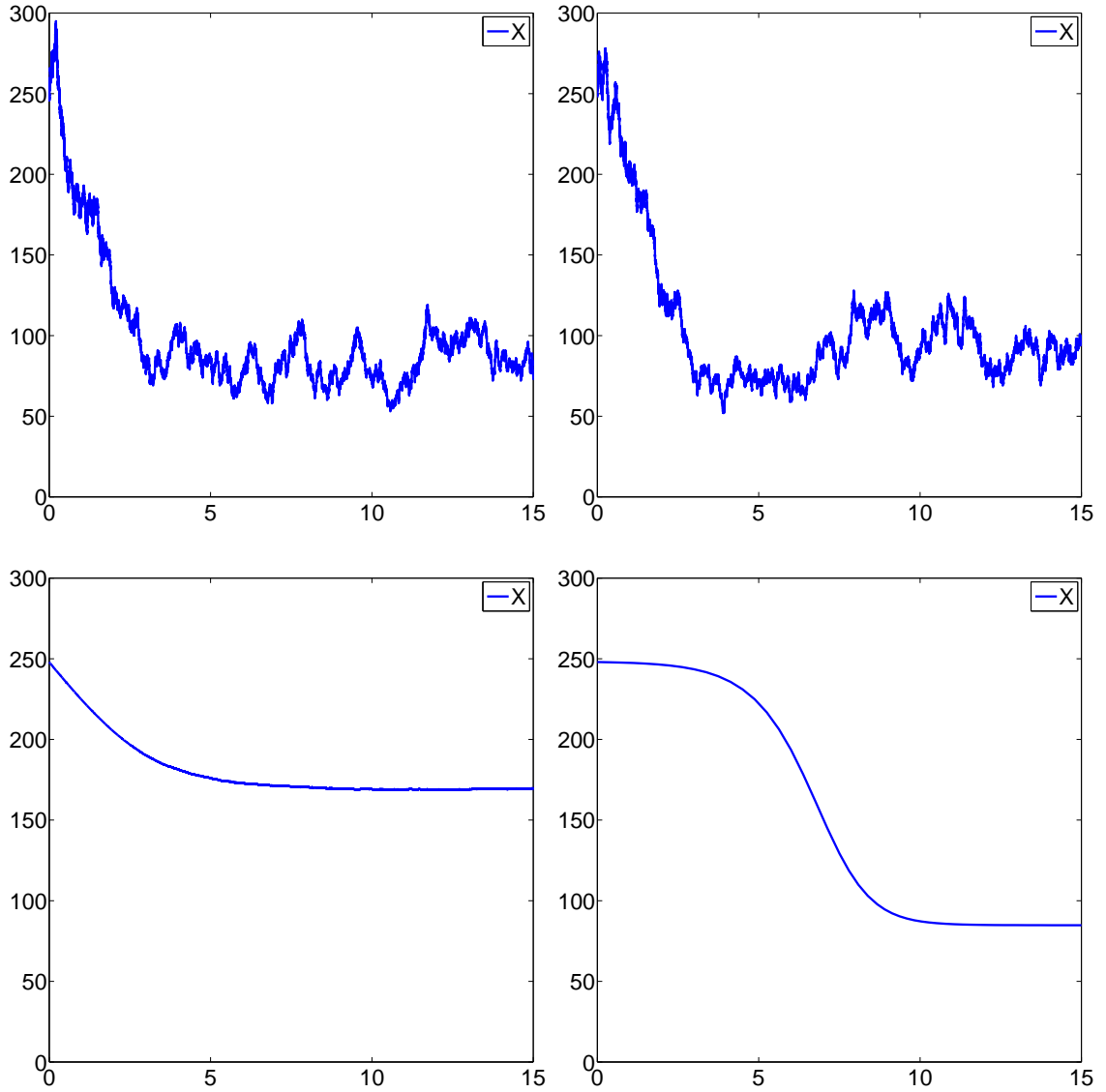
Figure 5.3: Integration of the Schlögl model over the interval $[0, 15]$ using SSA (top left), tau-leaping (top right), LLA/CLE (bottom left), and the default MARS solver for the RRE (bottom right). The vertical axis shows population amounts, while the horizontal axis shows simulation time. The value of τ utilized by tau-leaping and LLA/CLE was automatically determined by MARS.

It is interesting to note that the LLA/CLE method exhibits almost deterministic behaviour, and the lack of noise makes the initial downward trajectory shallower, and so the estimate it produces is much higher than the noisier SSA and tau-leaping methods.

With the same initial population values of $(10^5, 2 \times 10^5, 248)^T$, histograms of the populations at the final time were then produced by simulating 10,000 trajectories. The same methods were used, with the exception of GPU generation of trajectories being substituted for RRE solver, which would be ineligible for this test. Again, the RRE model was not included as it predicts the dynamics of the biochemical system deterministically.

The results are in the following Figure 5.4.

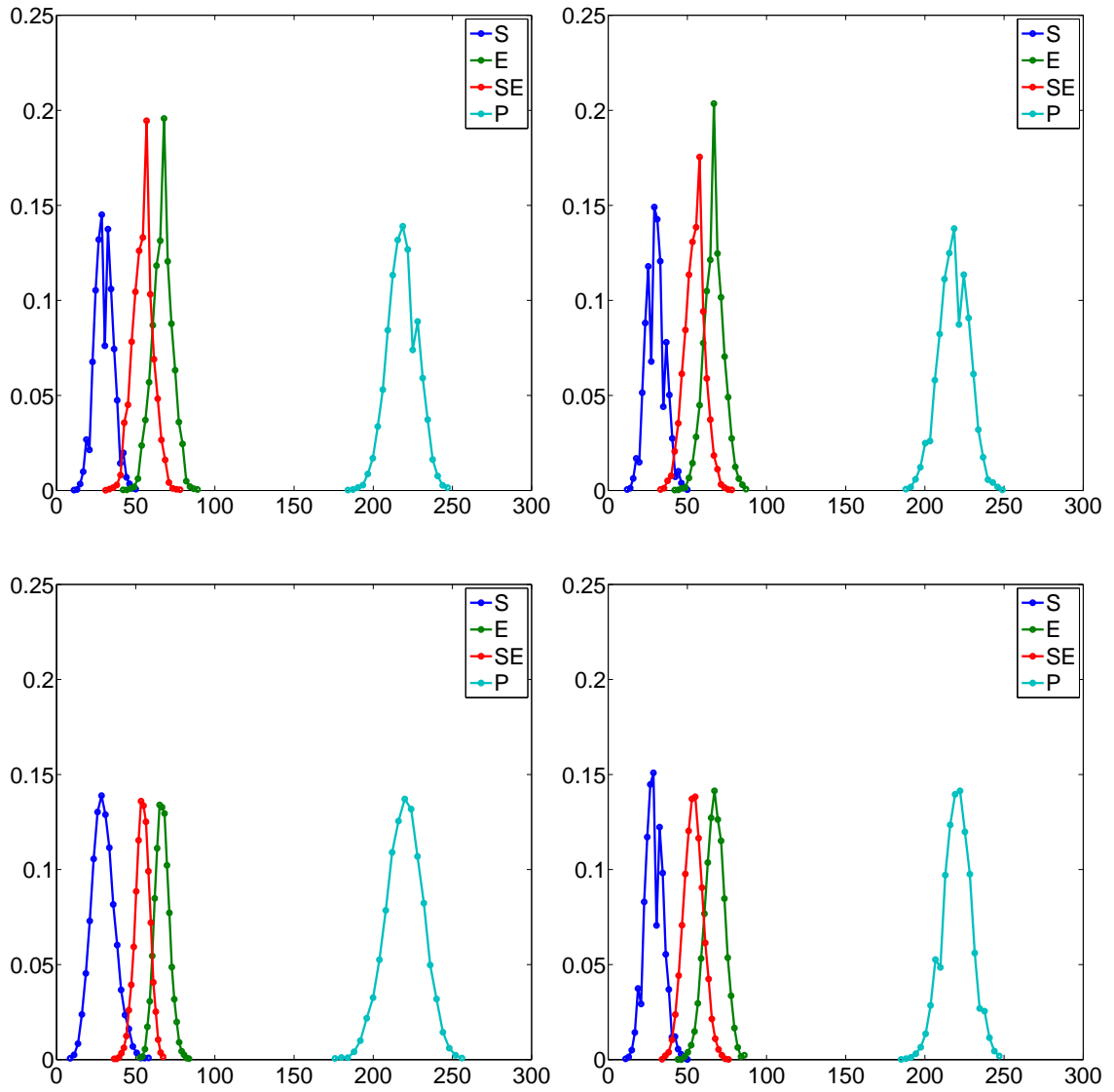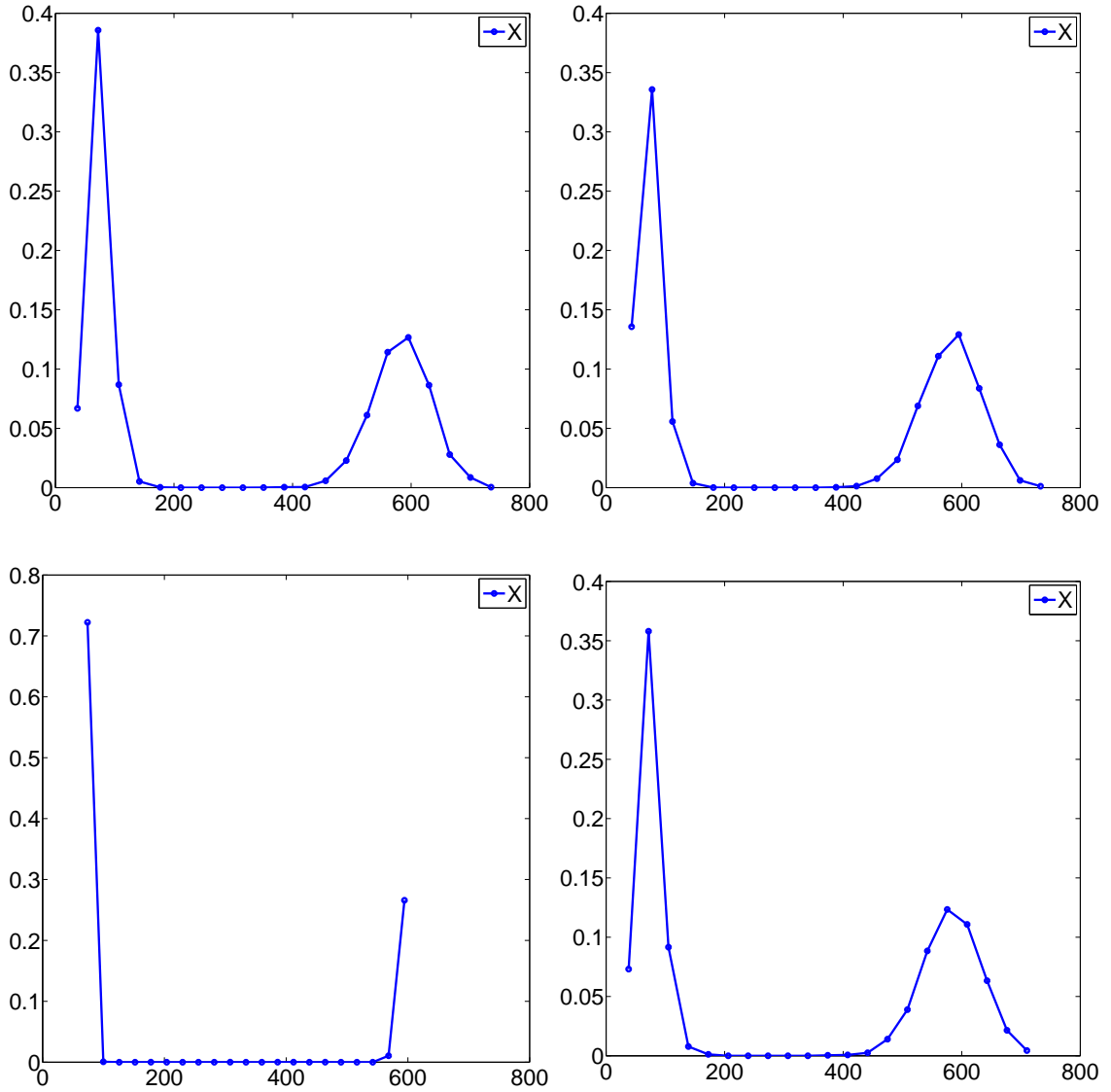Figure 5.4: Histogram of population amounts of 10,000 trajectories at the final time $T = 15$, after simulation of the Schlögl model over $[0, 15]$ using SSA (top left), tau-leaping (top right), LLA/CLE (bottom left), and GPU (bottom right). The vertical axis shows frequency, the horizontal axis shows final population amounts. The value of τ utilized by tau-leaping and LLA/CLE was automatically determined by MARS.

Of note, the SSA, tau-leaping, LLA/CLE trajectory generations took about 3.11×10^3 s, 9.68×10^2 s, and 3.26×10^2 s, respectively, to complete, while the the GPU implementation took only about 6.46 s to complete. This represents a significant speedup of about **480**

times over the SSA, **150** times over tau-leaping, and **50** times over LLA/CLE.

5.2.1 Varying the Initial Values in the Schlögl Model

The aforementioned bifurcation behaviour exhibited by the Schlögl model is responsible for the bimodal distribution seen in Figure 5.4. This behaviour can most clearly be seen in Figure 5.5.
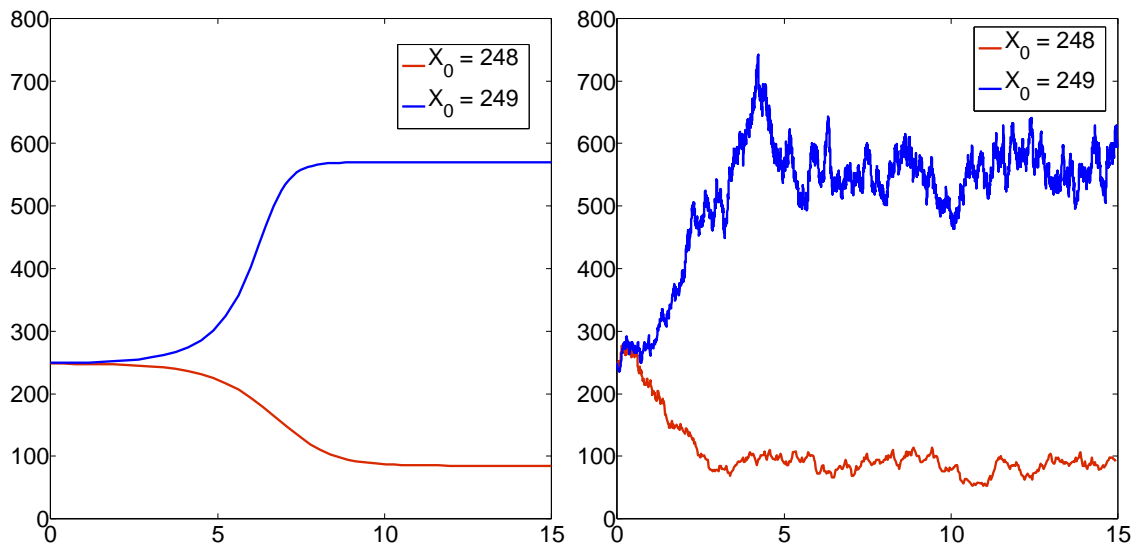
Figure 5.5: Population of species X in the Schlögl model over $[0, 15]$ with initial population set to 248 (bottom curve) or 249 (top curve). The right graph was generated using RRE and the, left using SSA for the CME.

5.3 Multilevel Monte Carlo Tau-leaping

The accuracy of the MLMC method, and the speedup obtained, as compared with the simulating of 10,000 trajectories using SSA were determined by direct comparison on three biochemical systems of interest. In each case, the M-value required by MLMC was determined through a mix of systematic testing and trial-and-error. It should be noted that the accuracy and the speed of the MLMC method is highly dependant on both this parameter and the tolerance specified by the user. An error of 10^{-2} and an M-value of 5 typically produced the best results.

5.3.1   Goldbeter-Koshland Switch

The Goldbeter-Koshland Switch model [12] consists of six species engaged in two reversible reactions (which are split into four reactions), and two non-reversible reactions. It models a phosphorylation-dephosphorylation system containing two enzymes, E_1 and E_2.

The relevant model data is summarized in Table 5.3.

	Reactions	Propensities	Reaction rates
R_1	$S + E_1 \xrightarrow{c_1} C_1$	$a_1(\mathbf{x}) = c_1 S E_1$	$c_1 = 0.05$
R_2	$C_1 \xrightarrow{c_2} S + E_1$	$a_2(\mathbf{x}) = c_2 C_1$	$c_2 = 0.1$
R_3	$C_1 \xrightarrow{c_3} P + E_1$	$a_3(\mathbf{x}) = c_3 C_1$	$c_3 = 0.1$
R_4	$P + E_2 \xrightarrow{c_4} C_2$	$a_4(\mathbf{x}) = c_4 P E_2$	$c_4 = 0.01$
R_5	$C_2 \xrightarrow{c_5} P + E_2$	$a_5(\mathbf{x}) = c_5 C_2$	$c_5 = 0.1$
R_6	$C_2 \xrightarrow{c_6} S + E_2$	$a_6(\mathbf{x}) = c_6 C_2$	$c_6 = 0.1$

Table 5.3: Goldbeter-Koshland Switch model

The model was integrated with with initial population values for $(S, E_1, C_1, P, E_2, C_2)^T$ of $(110, 100, 30,$
$30, 100, 30)^T$. The MLMC algorithm was implemented with an M-value of 5 and a tolerance of 10^{-2}, requiring 20 single coarse trajectories at level l_0, and 100, 20 coupled trajectories at levels $l_0/l_1, l_1/L$ respectively. SSA was utilized to generate 10,000 trajectories, with average population values taken at 100 steps over the integration. The integration interval was $[0, 5]$.

We show the numerical results in Figure 5.6.
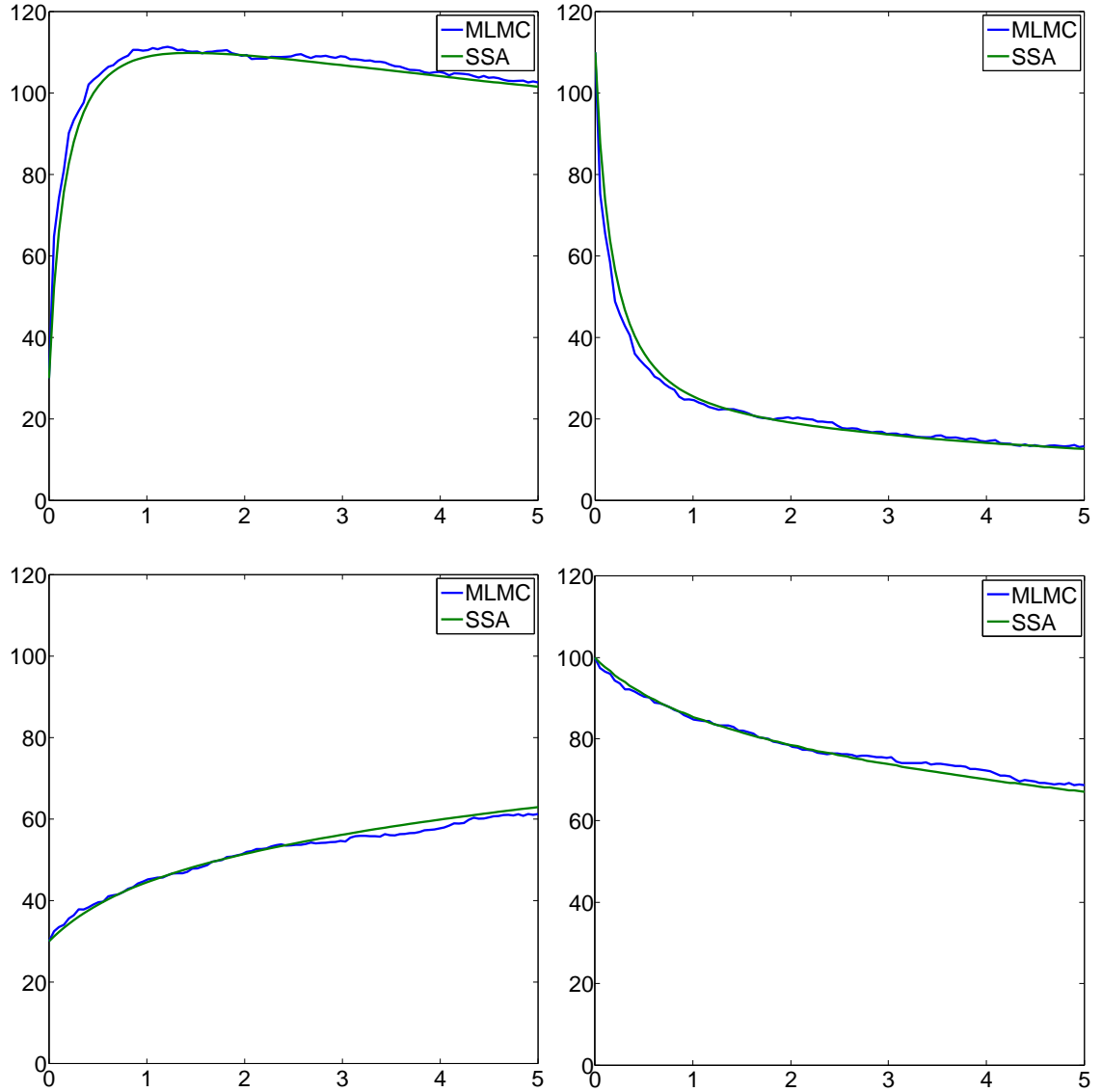
Figure 5.6:   Estimated average populations for the species in the Goldbeter-Koshland Switch model computed using the MLMC method compared to the average population values taken from 10,000 trajectories generated by the SSA. The results are presented for species C_1 (top left), S (top right), C_2 (bottom left), and E_2 (bottom right). The vertical axis shows population amounts, while the horizontal axis shows simulation time. Integration was taken over $[0, 5]$.

In addition to the MLMC plots visually showing an excellent agreement with the predictions of the exact SSA strategy, the method took about 2.55 seconds to complete,

while the 10,000 SSA trajectories took about 21.19 seconds. This represents a 8.31 times speedup. The precise errors from the final population values, are given in the Table 5.4.

Species	Absolute error	Relative error
S	6.55×10^{-1}	5.18×10^{-2}
E_1	1.05	3.67×10^{-2}
C_1	1.05	1.03×10^{-2}
P	1.44×10^{-2}	6.30×10^{-4}
E_2	1.69	2.51×10^{-2}
C_2	1.69	2.68×10^{-2}

Table 5.4: Errors for the estimated species populations in the Goldbeter-Koshland Switch model using the MLMC method compared to 10,000 SSA trajectories

5.3.2 Cyclical Reaction System

The cyclical reaction system model [12] consists of three species engaged three non-reversible reactions, forming a loop.

The relevant model data is presented in Table 5.5:

	Reactions	Propensities	Reaction rates
R_1	$A_1 \xrightarrow{c_1} A_2$	$a_1(\mathbf{x}) = c_1 A_1$	$c_1 = 0.1$
R_2	$A_2 \xrightarrow{c_2} A_3$	$a_2(\mathbf{x}) = c_2 A_2$	$c_2 = 0.1$
R_3	$A_3 \xrightarrow{c_3} A_1$	$a_3(\mathbf{x}) = c_3 A_3$	$c_3 = 0.1$

Table 5.5: Cyclical Reaction System model

The initial population values for $(A_1, A_2, A_3)^T$ are $(100, 80, 100)^T$. The MLMC algorithm was implemented with an M-value of 5 and a tolerance of 10^{-2}, requiring 1000 single coarse trajectories at level l_0, and 4, 4 coupled trajectories at levels $l_0/l_1, l_1/L$ respectively. The SSA was employed to simulate 10,000 trajectories, with average population values taken at 100 steps over the integration interval. The integration was performed over the time interval $[0, 20]$.

The numerical results are in Figure 5.7:
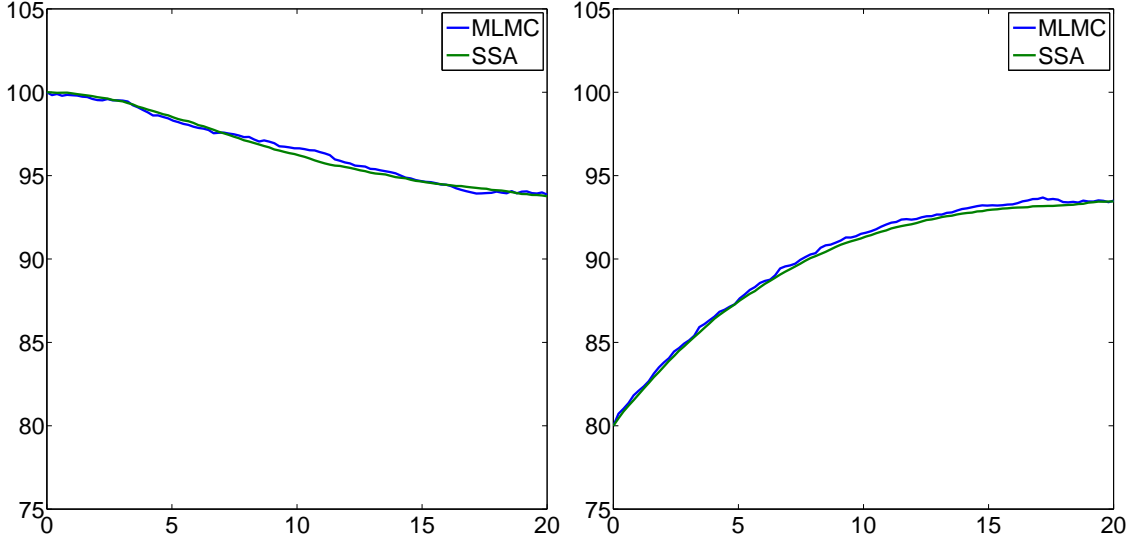
Figure 5.7: Evolution of the estimated average populations for species in the cyclical reaction system computed using MLMC compared to that for the average population values taken from 10,000 trajectories generated with the SSA. The results are for the species A_1 (left), A_2 (right). The vertical axis shows population amounts, while the horizontal axis shows simulation time. Integration was taken over $[0, 20]$.

We remark that the MLMC plots again show an excellent agreement with the results obtained for the 10,000 SSA trajectories. In addition the method took about 2.59 seconds to complete, while the 10,000 SSA trajectories took about 25.85 seconds. This represents a 9.98 times speedup. The precise errors from the final population values are in Table 5.6.

Species	Absolute error	Relative error
A_1	9.86×10^{-2}	1.05×10^{-3}
A_2	4.89×10^{-2}	5.23×10^{-4}
A_3	1.48×10^{-1}	1.59×10^{-3}

Table 5.6: Errors for the estimated species populations in the cyclical reaction system model using MLMC compared to 10,000 SSA trajectories

44

5.3.3   Potassium Channel

The Potassium Channel model [12] consists of five species subject to 5 reversible reactions (which are split into 10 simple reactions). Three species represent separate closed states, one species is an open state, and the last is an inactivation state.

The relevant model data is included in Table 5.7:

	Reactions	Propensities	Reaction rates
R_1	$C_1 \xrightarrow{c_1} C_2$	$a_1(\mathbf{x}) = c_1 S C_1$	$c_1 = 0.1$
R_2	$C_2 \xrightarrow{c_2} C_1$	$a_2(\mathbf{x}) = c_2 C_2$	$c_2 = 0.1$
R_3	$C_2 \xrightarrow{c_3} C_3$	$a_3(\mathbf{x}) = c_3 C_2$	$c_3 = 0.1$
R_4	$C_3 \xrightarrow{c_4} C_2$	$a_4(\mathbf{x}) = c_4 C_3$	$c_4 = 0.1$
R_5	$C_3 \xrightarrow{c_5} O$	$a_5(\mathbf{x}) = c_5 C_3$	$c_5 = 0.1$
R_6	$O \xrightarrow{c_6} C_3$	$a_6(\mathbf{x}) = c_6 O$	$c_6 = 0.1$
R_7	$O \xrightarrow{c_7} I$	$a_7(\mathbf{x}) = c_7 C_2$	$c_7 = 0.1$
R_8	$I \xrightarrow{c_8} O$	$a_8(\mathbf{x}) = c_8 C_3$	$c_8 = 0.1$
R_9	$I \xrightarrow{c_9} C_3$	$a_9(\mathbf{x}) = c_9 C_3$	$c_9 = 0.1$
R_{10}	$C_3 \xrightarrow{c_{10}} I$	$a_{10}(\mathbf{x}) = c_{10} O$	$c_{10} = 0.1$

Table 5.7: Potassium Channel model

The initial population values for the species $(C_1, C_2, C_3, O, I)^T$ are $(100, 50, 100, 50, 100)^T$. The MLMC technique was implemented with an M-value of 5 and a tolerance of 10^{-2}, requiring 1000 single coarse trajectories at level l_0, and 4, 4 coupled trajectories at levels $l_0/l_1, l_1/L$ respectively. SSA was applied to generate 10,000 trajectories, with average population values taken at 100 steps over the integration. The system was integrated over the time interval $[0, 10]$.

The results of our simulation using MARS are shown in Figure 5.8.
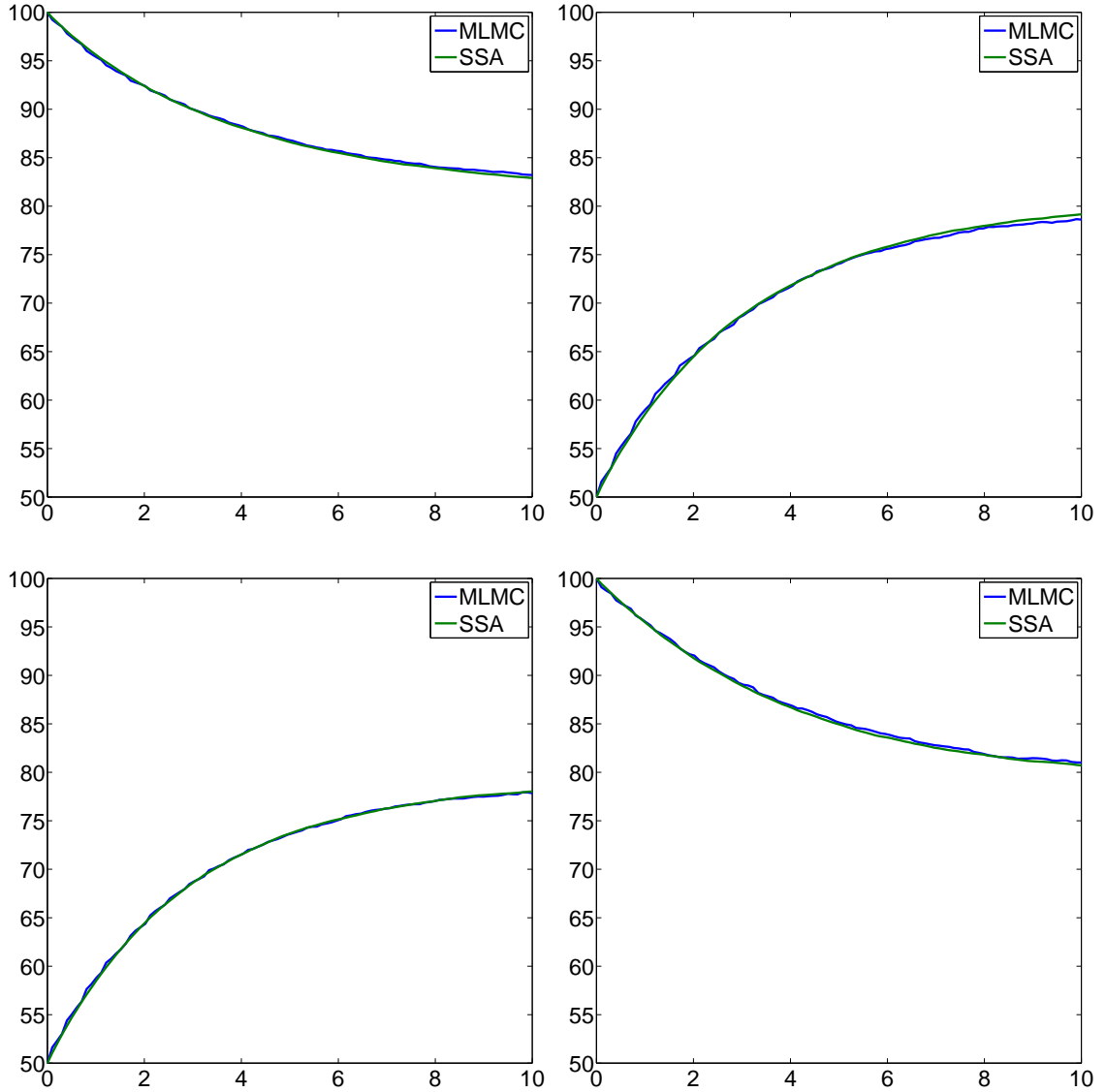
Figure 5.8: The evolution of estimated average populations for species in Potassium Channel model computed using MLMC compared to average population values taken from 10,000 SSA trajectories. The results are shown for species C_1 (top left), C_2 (top right), O (bottom left), and I (bottom right). The vertical axis shows population amounts, while the horizontal axis shows simulation time. Integration was taken over $[0, 10]$

As with the previous models, the numerical results for the Potassium Channel model obtained with the MLMC method and with 10,000 SSA trajectories show a very good agreement. Moreover, the method MLMC strategy took about 2.92 seconds to complete, while the 10,000 SSA trajectories took about 42.65 seconds. This represents a 14.61 times speedup. The precise errors for the final population value are given in the Table 5.8:

Species	Absolute error	Relative error
C_1	3.20×10^{-1}	3.58×10^{-3}
C_2	5.51×10^{-1}	6.96×10^{-3}
C_3	1.13×10^{-1}	1.42×10^{-3}
O	1.74×10^{-1}	2.23×10^{-3}
I	2.93×10^{-1}	3.63×10^{-3}

Table 5.8: Errors for the estimated species populations in the Potassium Channel model using MLMC compared to 10,000 SSA trajectories

Chapter 6

Conclusion

We have now analyzed the reasons for the development and implementation of stochastic methods for simulating systems of biochemical reactions. Understanding these systems is crucial to the study of organism functionality and behaviour, and so methods that allow fast simulation and predictive capabilities are valuable to the field of Systems Biology, and by extension Biology as a whole. We have introduced the various traditional approaches for modelling biochemical systems, such as the Reaction Rate Equation, as well as the development of the Chemical Master Equation and in turn the Stochastic Simulation Algorithm, tau-leaping adaptations of Stochastic Simulation Algorithm, and the Chemical Langevin Equation. Further, we discussed the concept and implementation of the Multilevel Monte Carlo Tau-leaping method, showing that it approximates the solution to the Chemical Master Equation almost as accurately as the gold standard of 10,000 SSA trajectories, but with much reduced computational complexity.

Finally, we presented our software implementation making use of all these methods with a fair amount of capability, one which would allow anyone with an SBML file to obtain and plot data from systems of biochemical reactions with little knowledge of the mathematical techniques being used. This software is also scalable and parallalizable across either multiple CPUs or CPU cores, or a GPU, allowing for greatly reduced simulation times.

A priority area with regards to future enhancements to the software would be to implement parallelization on a GPU using Nvidia's CUDA GPU programming language. The current GPU implementation, working purely in MATLAB, has its limitations, namely in terms of flexibility. It does not allow the recording of systems states throughout the

simulation, nor currently provide an easy way to draw samples variable from a Poisson random variable. Additionally, many parameters must be hard-coded into the various function calls required, an obvious problem in dynamic software. Implementation using CUDA would solve nearly all of these problems with fewer or no workarounds, and given that it is a lower-level language, may also provide speed increases.

Implementation of adaptive tau-leaping procedures would also be a very useful addition. Adaptive tau-leaping uses a variable step size that can sidestep some of the problems explicit constant step tau-leaping procedures suffer from, primarily some populations being driven negative. In addition, adaptive strategies would be important when dealing with stiff systems, as an adaptive step-size would lend itself particularly well to simulating systems containing multiple time scales.

Moving forward, the modelling of systems of biochemical reaction will continue to be crucial to modern scientific progress. The importance of having the ability to predict and understand the behaviour of such systems cannot be overstated, and so it will likely be an area of relevance for some time to come.

Bibliography

[1] D.F. Anderson, D.J. Higham, 2012, Multilevel Monte Carlo for continuous time Markov Chains with application to biochemical kinetics, *SIAM Multiscale Modeling and Simulation*, **10**, 146–179.

[2] D.F. Anderson, D.J. Higham, Y. Sun, 2013, Complexity of Multilevel Monte Carlo Tau-Leaping, *arXiv:1310.2676v1 [math.NA]*.

[3] B.J. Bornstein, S.M. Keating, A. Jouraku, M. Hucka, 2008, LibSBML: An API Library for SBML, *Bioinformatics*, **24.6**, 880–881.

[4] D.T. Gillespie, 1976, A general method for numerically simulating the stochastic time evolution of coupled chemical reactions, *Journal of Computational Physics*, **22**, 403–434.

[5] D.T. Gillespie, 2007, Stochastic Simulation of Chemical Kinetics, *Annual Review of Physical Chemistry*, **58**, 35–55.

[6] D.J. Higham, 2001, An Algorithmic Introduction to Numerical Simulation of Stochastic Differential Equations, *SIAM Review*, **43.3**, 525–546.

[7] D.J. Higham, 2008, Modeling and Simulating Chemical Reactions, *SIAM Review*, **50.2**, 347–368.

[8] S. Ilie, W.H. Enright, K.R. Jackson, 2009, Numerical solution of stochastic models of biochemical kinetics, *Canadian Applied Mathematics Quarterly*, **17.3**, 523–554.

[9] S.M. Keating, B.J. Bornstein, A. Finney, M. Hucka, 2006, SBMLToolbox: an SBML toolbox for MATLAB users, *Bioinformatics*, **22.10**, 1275–1277.

[10] H.H. McAdams, A. Arkin, 1997, Stochastic mechanisms in gene expression, *Proceedings of the National Academy of Sciences of the United States of America*, **94**, 814-819.

[11] D. McQuarrie, 1967, Stochastic approach to chemical kinetics, *Journal of Applied Probability*, **4**, 413–478.

[12] B. Mélykúti, K. Burrage, K.C. Zygalakis, 2010, Fast stochastic simulation of biochemical systems by alternative formulations of the chemical Langevin equation, *The Journal of Chemical Physics*, **132**, 164109.

[13] M. Rathinam, L.R. Petzold, Y. Cao, D.T. Gillespie, 2003, Stiffness, Stochastic Chemically Reacting Systems: The Implicit Tau-Leaping Method, *Journal of Chemical Physics*, **119**, 12784.

[14] D.J. Wilkinson, 2009, Stochastic modelling for quantitative description of heterogeneous biological systems, *Nature Reviews Genetics*, **10.2**, 122–133.

Appendix A

Source Code

Please note the source code can be downloaded in its entirety, including with the required libraries, from `https://github.com/dbarrows/mars`, though it may not yet be available for download at the time of publication. It should be noted that the source code download also contains the SBMLToolbox and libSBML, libraries and tools required to interact with SBML files. The source code included in this appendix does *not* include these files, it contains only the original work of the author, which is limited to components that build on top of these additional libraries and tools.

```matlab
 1 % Generates a file 'calculatePropensities.m' that will contain a function able to calculate↵
   the necessary propensities for that system.
 2 function GeneratePropensityCalculatorFile(SBMLModel, VHolder)
 3
 4 % determine number of reactions and species present in the model
 5 numReactions = length(SBMLModel.reaction);
 6 numSpecies = length(SBMLModel.species);
 7
 8 [cNames, cValues] = GetParameters(SBMLModel);
 9
10 % matricies to hold propensity values
11 A = zeros(numReactions,1);
12
13 V = VHolder.V;
14 vNumOfReactant = VHolder.vNumOfReactant;
15 vReactant = VHolder.vReactant;
16 vDimerMap = VHolder.vDimerMap;
17
18 fid = fopen('calculatePropensities.m','w');
19 fprintf(fid,'function A = calculatePropensities(X)\n\n');
20
21 for i = 1:numReactions
22
23     % set initially to that reaction's parameter
24     format long;
25     fprintf(fid, 'A(%d) = (%d)', i, cValues(i) );
26
27     % multiply current value by each ractant's value if applicable, and account for↵
   dimerisation reactions
28     for k = 1:(vNumOfReactant(i));
29
30         curSpeciesIndex = vReactant(i,k);
31
32         fprintf(fid,'*X(%d)', curSpeciesIndex);
33
34         % determine is reactant is part of a dimerisation reaction using dimerisation map,↵
   then alter propensity accordingly
35         dimer_number = vDimerMap(curSpeciesIndex, i);
36         if dimer_number > 1
37             count = 1;
38             while (count < dimer_number)
39                 fprintf(fid,'*(X(%d)-%d)', curSpeciesIndex, count);
40                 count = count + 1;
41             end
42
43             fprintf(fid,'/%d', factorial(dimer_number) );
44         end
45     end
46
47     fprintf(fid, ';\n', i, cValues(i) );
48 end
49
50 fprintf(fid,'\nend',numReactions);
51
52 fclose(fid);
53
54 end
```

```matlab
1  function totalsIndecies =  GenerateSpecifiedTotalsCalculatorFile(SBMLModel)
2
3  numReactions = length(SBMLModel.reaction);
4  numSpecies = length(SBMLModel.species);
5
6  [speNames, speValues] = GetSpecies(SBMLModel);
7
8  totalsIndecies = zeros(numSpecies,1);
9
10 fid = fopen('calculateSpecifiedTotals.m','w');
11 fprintf(fid,'function X = calculateSpecifiedTotals(X)\n\n');
12
13 count = 0;
14 for i = 1:length(SBMLModel.rule)
15     curRule = SBMLModel.rule(i);
16
17     for j = 1:length( speNames )
18         if strcmp( speNames(j), curRule.variable )
19
20             totalsIndecies(count+1) = j;
21             count = count + 1;
22             fprintf(fid, 'X(%d) = 0', j);
23
24             % token string array
25             ruleToks = strsplit( curRule.formula ,'+');
26
27             for l = 1:length( ruleToks )
28                 for m = 1:length(speNames)
29                     if strcmp( speNames(m), ruleToks(l) )
30                         fprintf(fid, ' + X(%d)', m);
31                     end
32                 end
33             end
34
35             fprintf(fid, ';\n', m);
36
37             break;
38         end
39     end
40 end
41
42 fprintf(fid,'\nend',numReactions);
43
44 fclose(fid);
45
46 totalsIndecies = totalsIndecies(1:count);
47
48 end
```

```matlab
1  function speConstIndecies = GetConstantSpeciesIndecies(SBMLModel)
2
3  numSpecies = length(SBMLModel.species);
4
5  speConstIndecies = zeros(numSpecies,1);
6
7  count = 0;
8  for i = 1:numSpecies
9      if SBMLModel.species(i).constant
10         speConstIndecies(count + 1) = i;
11         count = count + 1;
12     end
13 end
14
15 speConstIndecies = speConstIndecies(1:count,1);
16
17 end
```

```matlab
 1 function [cNames, cValues] = GetParameters(SBMLModel)
 2
 3 numReactions = length(SBMLModel.reaction);
 4
 5 % get all parameter names, values
 6 [allCNames, allCValues] = GetAllParameters(SBMLModel);
 7
 8 % create matricies to hold parameter names, values
 9 cValues = zeros(numReactions,1);
10 cNames = cell(numReactions,1);
11
12 % get parameters for each individual reaction as they may not be in order
13 for i = 1:numReactions
14
15     % attempt to get parameters from local rection context
16     [cNamesTemp,cValuesTemp] = GetParameterFromReaction(SBMLModel.reaction(i));
17
18     % if parameter values are NOT embedded in each local reaction context (return values will↙
be NULL)
19     if ( length(cNamesTemp) == 0 ) && ( length(cValuesTemp) == 0 )
20
21         % declare parameter found flag, get string from reaction formula field, tokenize it by↙
operator
22         done = 0;
23         kinLawString = SBMLModel.reaction(i).kineticLaw.formula;
24         kinLawStringToks = strsplit( kinLawString ,'*');
25
26         % searches for each token in the list of all parameters in the model, assigns those↙
values to the correct reaction if found
27         for j = 1:length(kinLawStringToks)
28             curTok = kinLawStringToks(j);
29             for k = 1:length(allCNames);
30                 if strcmp( allCNames{k} , curTok )
31                     cNames{i} = allCNames{k};
32                     cValues(i) = allCValues(k);
33                     done = 1;
34                     break;
35                 end
36
37                 if done == 1
38                     break;
39                 end
40             end
41
42             if done == 1
43                 break;
44             end
45         end
46     % if parameters were embedded in the local reaction contexts, assign them
47     else
48         cNames(i) = cNamesTemp;
49         cValues(i) = cValuesTemp;
50     end
51 end
52
53 end
```

```matlab
1 function GraphHist(Y, speciesToGraph, speNames, split_flag, filename, method_name)
2
3
4 num_bins = length(Y)^(1/3);
5
6 specific_species_flag = 0;
7 speIndLength = length(speciesToGraph);
8
9 if speIndLength == 0
10     end_point = min ( size(Y) );
11 else
12     end_point = speIndLength;
13     specific_species_flag = 1;
14 end
15
16 if ~split_flag
17     figure
18     hold all
19 end
20
21 for i = 1:end_point
22
23     if specific_species_flag
24         current_species = speciesToGraph(i);
25     else
26         current_species = i;
27     end
28
29
30     min_val = min( Y(:,current_species) );
31     max_val = max( Y(:,current_species) );
32
33     % determine optimal number of bins (educated guess)
34     space = max_val - min_val + 1;
35     if space > num_bins
36         bins = linspace(min_val, max_val, num_bins);
37         h = hist( Y(:,current_species) , num_bins );
38     else
39         bins = linspace(min_val, max_val, space);
40         h = hist( Y(:,current_species) , space );
41     end
42
43     if split_flag
44         figure
45     end
46
47     plot( bins, h/length(Y) , 'o-' );
48
49     if split_flag
50         legend( speNames(current_species) );
51         xlabel('Number of Species','FontSize',12, 'FontName', 'Helvetica');
52         ylabel('Frequency','FontSize',12,'FontName', 'Helvetica');
53         title('Histogram of number of species at end of simulation','FontSize',16,'FontName',↙
'Helvetica');
54     end
55 end
56
57 if ~split_flag
```

```
58      hold off
59
60      if specific_species_flag
61          legend( speNames(speciesToGraph) );
62      else
63          legend( speNames )
64      end
65
66      xlabel('Number of Species','FontSize',12, 'FontName', 'Helvetica');
67      ylabel('Frequency','FontSize',12,'FontName', 'Helvetica');
68
69      title_string = ['Histogram of number of species at end of simulation from model source '''↵
filename ''' using ' method_name];
70      title(title_string, 'Fontsize', 16,'FontName', 'Helvetica');
71 end
72
73 end
```

```matlab
 1 function GraphResults(Y, time, speciesToGraph, speNames, split_flag, filename, method_name)
 2
 3 specific_species_flag = 0;
 4 speIndLength = length(speciesToGraph);
 5
 6 if speIndLength == 0
 7     dims = size(Y);
 8     end_point = dims(1);
 9 else
10     end_point = speIndLength;
11     specific_species_flag = 1;
12 end
13
14 if ~split_flag
15     figure
16     hold all
17 end
18
19 for i = 1:end_point
20
21     if specific_species_flag
22         current_species = speciesToGraph(i);
23     else
24         current_species = i;
25     end
26
27     if split_flag
28         figure
29     end
30
31     plot( time , Y(current_species,:) );
32
33     if split_flag
34         legend( speNames(current_species) );
35         xlabel('Time','FontSize',12, 'FontName', 'Helvetica');
36         ylabel('Number of Species','FontSize',12,'FontName', 'Helvetica');
37         title('Species vs time using SSA','FontSize',16,'FontName', 'Helvetica');
38     end
39 end
40
41 if ~split_flag
42     hold off
43
44     if specific_species_flag
45         legend( speNames(speciesToGraph) );
46     else
47         legend( speNames )
48     end
49
50     xlabel('Time','FontSize',12, 'FontName', 'Helvetica');
51     ylabel('Number of Species','FontSize',12,'FontName', 'Helvetica');
52
53     title_string = ['Species vs time from model source ''' filename ''' using ' method_name];
54     title(title_string,'FontSize',16,'FontName', 'Helvetica');
55 end
56
57 end
```

```matlab
 1 function varargout = MARS(filename, varargin)
 2
 3 % Options:
 4 %
 5 % 'Hist'    - generate a histogram of the results of 10,000 trajecories
 6 % 'Verbose' - enable diagnostic outputs
 7 % 'Time'    - max time to run the simulation for
 8 % 'Record'  - step size between recording simulation state information
 9 % 'Tau'     - use tau-leaping
10 % 'CLE'     - use Chemical Langevin Equation
11 % 'RRE'     - use raction rate equations
12 % 'GPU'     - use the system's CUDA-supported GPU for multiple trajectory generation
13 % 'MLMC'    - generate histogram using Multi-level Monte-Carlo simulation (experimental)
14 % 'Error'   - error to use for MLMC method
15 % 'Steps'   - number of data points to generate over the integration interval for all non-GPU↵
parallel methods
16 % 'Graph'   - plot a graph of the results
17
18 % default values for user-provided arguments
19 hist_flag      = 0;
20 verbose_flag   = 0;
21 tau_flag       = 0;
22 cle_flag       = 0;
23 rre_flag       = 0;
24 stiff_flag     = 0;
25 split_flag     = 0;
26 keep_flag      = 0;
27 record_flag    = 0;
28 gpu_flag       = 0;
29 graph_flag     = 0;
30 mlmc_flag      = 0;
31 m_flag         = 0;
32 err_flag       = 0;
33 steps_flag     = 0;
34 tfinal         = 50;
35 recordStep     = 20;
36 numSteps       = 100;
37 err            = 0;
38 speciesToGraph = [];
39
40 i = 1;
41 while (1+i) <= nargin
42     switch varargin{i}
43         case 'Hist'
44             hist_flag = 1;
45
46         case 'Verbose'
47             verbose_flag = 1;
48
49         case 'Time'
50             if (i+2) > nargin
51                 disp( sprintf('\nSimulation time argument missing.\n') );
52                 return;
53             else
54                 i = i + 1;
55                 time_arg = varargin{i};
56                 if ~isnumeric(time_arg)
57                     disp( sprintf('\nSimulation time argument must be a number.\n') );
```

```matlab
 58                                return;
 59                            else
 60                                tfinal = time_arg;
 61                            end
 62                    end
 63
 64            case 'Record' % get record step argument, check for validity (integer)
 65                if (i+2) > nargin
 66                    disp( sprintf('\nRecord step size argument missing.\n') );
 67                    return;
 68                else
 69                    i = i + 1;
 70                    record_arg = varargin{i};
 71                    if ~isnumeric(record_arg) || mod(record_arg,1) ~= 0
 72                        disp( sprintf('\nRecord step size argument must be an integer.\n') );
 73                        return;
 74                    else
 75                        recordStep = record_arg;
 76                        record_flag = 1;
 77                    end
 78                end
 79
 80            case 'Tau' % use tau-leaping, get value to use for tau
 81                if (i+2) > nargin
 82                    tau = 0;
 83                else
 84                    next_arg = varargin{i+1};
 85                    if isnumeric(next_arg)
 86                        tau = next_arg;
 87                        i = i + 1;
 88                    else
 89                        tau = 0;
 90                    end
 91                end
 92                tau_flag = 1;
 93
 94            case 'CLE' % use Langevin leaping algorithm, get value to use for tau
 95                if (i+2) > nargin
 96                    tau = 0;
 97                else
 98                    next_arg = varargin{i+1};
 99                    if isnumeric(next_arg)
100                        tau = next_arg;
101                        i = i + 1;
102                    else
103                        tau = 0;
104                    end
105                end
106                cle_flag = 1;
107
108            case 'MLMC' % use MLMC method, get value to use for M, defaults to 100 steps
109                if (i+2) > nargin
110                    M = 0;
111                else
112                    m_arg = varargin{i+1};
113                    if isnumeric(m_arg) && mod(m_arg,1) ~= 0
114                        disp( sprintf('\nMLMC M-value argument must be an integer.\n') );
115                        return;
```

```matlab
116                 elseif isnumeric(m_arg) && mod(m_arg,1) == 0
117                     M = m_arg;
118                     i = i + 1;
119                     m_flag = 1;
120                 else
121                     M = 0;
122                 end
123             end
124             mlmc_flag = 1;
125
126         case 'Steps' % get specific numer of steps to generate for parallel methods not on a↙
GPU
127             if (i+2) > nargin
128                 disp( sprintf('\nNumber of steps size argument missing.\n') );
129                 return;
130             else
131                 step_arg = varargin{i+1};
132                 if isnumeric(step_arg) && mod(step_arg,1) ~= 0
133                     disp( sprintf('\nNumber of steps argument must be an integer.\n') );
134                     return;
135                 elseif isnumeric(step_arg) && mod(step_arg,1) == 0
136                     numSteps = step_arg;
137                     i = i + 1;
138                 else
139                     disp( sprintf('\nNumber of steps size argument missing or invalid.\n') );
140                     return;
141                 end
142             end
143
144         case 'Error' % get error to use for MLMC method, overrides default
145             if (i+2) > nargin
146                 disp( sprintf('\nError argument missing.\n') );
147                 return;
148             else
149                 i = i + 1;
150                 err_arg = varargin{i};
151                 if ~isnumeric(err_arg)
152                     disp( sprintf('\nError argument missing.\n') );
153                     return;
154                 elseif isnumeric(err_arg) && err_arg < 0
155                     disp( sprintf('\nError must be a positive number.\n') );
156                 else
157                     err = err_arg;
158                 end
159             end
160
161         case 'Graph' % whether or not to grapht the results, and if so for which species↙
(default is all)
162             if (i+2) > nargin
163                 speciesToGraph = [];
164             else
165                 next_arg = varargin{i+1};
166                 if isvector(next_arg) && isnumeric(next_arg)
167                     speciesToGraph = next_arg;
168                     i = i + 1;
169                 else
170                     speciesToGraph = [];
171                 end
```

```
172                 end
173                 graph_flag = 1;
174
175         case 'RRE'
176                 rre_flag = 1;
177
178         case 'Stiff'
179                 stiff_flag = 1;
180
181         case 'Split'
182                 split_flag = 1;
183
184         case 'Keep'
185                 keep_flag = 1;
186
187         case 'GPU'
188                 gpu_flag = 1;
189
190         otherwise
191                 disp( sprintf('\nInvalid option detected at argument %d.\n',i) );
192                 return;
193     end
194     i = i + 1;
195 end
196 if (tau_flag + cle_flag + rre_flag + mlmc_flag + gpu_flag) > 1
197     disp( sprintf('\nInvalid options: multiple methods selected. You may only pick one of↵
CLE, Tau-Leaping, RRE, MLMC, or GPU\n') );
198     return
199 end
200
201 if tfinal == 0
202     disp( sprintf('\nInvalid final time argument\n') );
203     return
204 end
205
206 if recordStep == 0
207     disp( sprintf('\nInvalid record step size argument\n') );
208     return
209 end
210
211 if numSteps == 0
212     disp( sprintf('\nInvalid number of steps argument\n') );
213     return
214 end
215
216 if m_flag && M < 2
217     disp( sprintf('\nMLMC M-value must be an integer greater than 1\n') );
218     return
219 end
220
221 addpath(genpath('./toolbox/SBMLToolbox'));
222
223 platform_str = computer;
224
225 % get type of platform so proper libraries can be added to path, currently only PC, Mac↵
supported
226 switch platform_str
227     case 'MACI64'
```

```matlab
228            addpath(genpath('./toolbox/libSBML/mac'));
229      case 'PCWIN'
230            addpath(genpath('./toolbox/libSBML/win32'));
231      case 'PCWIN64'
232            addpath(genpath('./toolbox/libSBML/win64'));
233      case 'GLNXA64'
234            addpath(genpath('./toolbox/libSBML/linux'));
235      otherwise
236            disp('Platform not supported');
237            return;
238 end
239
240 % create files to be filled, then close all
241 file_list = {'calculatePropensities.m';...
242              'calculateSpecifiedTotals.m';...
243              'RRE_functions.m';...
244              'fireGpuTrajectories.m'};
245
246 num_files = length(file_list);
247 for i = 1:num_files
248     cur_file = file_list{i};
249     fid = fopen(cur_file,'w');
250     fclose(fid);
251 end
252
253 % get system (model) inforamtion from SBML file
254 SysInf = SSA_setup(filename, verbose_flag);
255 numSpecies        = SysInf.numSpecies;
256 numReactions      = SysInf.numReactions;
257 speNames          = SysInf.speNames;
258 speValues         = SysInf.speValues;
259 cNames            = SysInf.cNames;
260 cValues           = SysInf.cValues;
261 speConstIndecies  = SysInf.speConstIndecies;
262 totalsIndecies    = SysInf.totalsIndecies;
263 VHolder           = SysInf.VHolder;
264
265 % check for bad species-to-graph entries
266 if graph_flag && ~mlmc_flag
267     numSpeToGraph = length(speciesToGraph);
268     if numSpeToGraph ~= 0
269         for i = 1:numSpeToGraph
270
271             curIndex = speciesToGraph(i);
272             if curIndex > numSpecies || curIndex < 1
273                 disp( sprintf(['Error: invalid species index provided, exceeds number of↙
species in system or is less than 1.'...
274                               ' Graph will not be displayed.\n']) );
275                 graph_flag = 0;
276             end
277
278             for j = 1:(i-1)
279                 checkIndex = speciesToGraph(j);
280                 if checkIndex == curIndex
281                     disp( sprintf('Error: invalid species index provided, duplicate index.↙
Graph will not be displayed.\n') );
282                     graph_flag = 0;
283                 end
```

```matlab
284              end
285
286          end
287      end
288 end
289
290 method_name = '';
291 rehash
292
293 if gpu_flag
294      Y = SSA_gpu(filename, SysInf, tfinal, verbose_flag);
295      method_name = 'SSA on GPU';
296
297 elseif mlmc_flag
298      %open parallel pool based on installed toolbox version
299      version_less_flag = verLessThan('distcomp', '6.3');
300      if version_less_flag
301          matlabpool open;
302      else
303          parpool;
304      end
305
306      [Mean, Step] = MLMCGen(SysInf, tfinal, numSteps, verbose_flag, split_flag,↵
speciesToGraph, graph_flag, M, err);
307      Y = Mean;
308      varargout{3} = Step;
309      time = linspace(0,tfinal,numSteps);
310      varargout{2} = time;
311      method_name = 'MLMC';
312
313      % close parallel pool
314      if version_less_flag
315          matlabpool close;
316      else
317          delete(gcp);
318      end
319
320 elseif hist_flag
321      %open parallel pool based on installed toolbox version
322      version_less_flag = verLessThan('distcomp', '6.3');
323      if version_less_flag
324          matlabpool open;
325      else
326          parpool;
327      end
328
329      % use indicated method to generate trajectories accross multiple CPUs or CPU cores
330      if tau_flag
331          [Y,Mean,Std] = SSAGen_parfor_tauleap(SysInf, tfinal, recordStep, verbose_flag, tau,↵
speciesToGraph, numSteps, graph_flag, 0);
332          method_name = 'SSA with tau-leaping on parallel CPUs';
333      elseif cle_flag
334          [Y,Mean,Std] = SSAGen_parfor_cle(SysInf, tfinal, recordStep, verbose_flag, tau,↵
speciesToGraph, numSteps, graph_flag);
335          method_name = 'CLE on parallel CPUs';
336      elseif rre_flag
337          disp( sprintf('\n''Hist'' is not a valid option to use with the Reaction Rate↵
Equation method\n') );
```

```matlab
338        else
339            [Y,Mean,Std] = SSAGen_parfor(SysInf, tfinal, recordStep, verbose_flag,↵
speciesToGraph, numSteps, graph_flag);
340            method_name = 'SSA on parallel CPUs';
341        end
342
343        varargout{2} = Mean;
344        varargout{3} = Std;
345
346        % close parallel pool
347        if version_less_flag
348            matlabpool close;
349        else
350            delete(gcp);
351        end
352
353    else
354        % generate single trajectory based on indicated method
355        if tau_flag
356            [time, Y] = SSAGen_tauleap(SysInf, tfinal, recordStep, verbose_flag, tau);
357            method_name = 'SSA with tau-leaping';
358        elseif cle_flag
359            [time, Y] = SSAGen_cle(SysInf, tfinal, recordStep, verbose_flag, tau);
360            method_name = 'CLE';
361        elseif rre_flag
362            if record_flag
363                disp(sprintf('\nWarning: ''Record'' argument will be ignored - not valid with↵
Reaction Rate Equation method\n'));
364            end
365            [time, Y] = RREGen(SysInf, tfinal, verbose_flag, stiff_flag);
366            method_name = 'RRE';
367        else
368            [time, Y] = SSAGen(SysInf, tfinal, recordStep, verbose_flag);
369            method_name = 'SSA';
370        end
371
372        varargout{2} = time;
373    end
374
375    varargout{1} = Y;
376
377    arg_type = class(filename);
378    switch arg_type
379        case 'struct'
380            filename = filename.name;
381    end
382
383    % graph end of simulation histogram is using a parallel method and graph flag has been set
384    if graph_flag
385        if gpu_flag || hist_flag
386            GraphHist(Y, speciesToGraph, speNames, split_flag, filename, method_name);
387        else
388            GraphPlot(Y, time, speciesToGraph, speNames, split_flag, filename, method_name);
389        end
390    end
391
392    % keep temporary files if indicated
393    if ~keep_flag
```

```
394        for i = 1:num_files
395            cur_file = file_list{i};
396            delete(cur_file);
397        end
398 end
399
400 end
```

```matlab
  1 function [Y, Step] = MLMCGen(SysInf, tfinal, numSteps, verbose_flag, split_flag,↵
speciesToGraph, graph_flag, M, err)
  2
  3 numSpecies         = SysInf.numSpecies;
  4 numReactions       = SysInf.numReactions;
  5 speNames           = SysInf.speNames;
  6 speValues          = SysInf.speValues;
  7 cNames             = SysInf.cNames;
  8 cValues            = SysInf.cValues;
  9 speConstIndecies   = SysInf.speConstIndecies;
 10 totalsIndecies     = SysInf.totalsIndecies;
 11 VHolder            = SysInf.VHolder;
 12
 13 % initial values
 14 X = speValues;
 15 numSteps = numSteps - 1;
 16
 17 if err == 0
 18     err = 1/100;
 19 end
 20
 21 % extract V from VHolder and display
 22 V = VHolder.V;
 23
 24 if verbose_flag
 25     disp( sprintf('Stoichiometric Matrix:\n') ); disp(V);
 26 end
 27
 28 % matricies to hold propensity values, number of species present after each step, and the↵
length of each step
 29 A = zeros(numReactions,1);
 30
 31 % parameters
 32 N = max(X);
 33
 34 % default M
 35 if M == 0
 36     M = 3;
 37 end
 38
 39 alp = zeros(numSpecies,1);
 40 for i = 1:numSpecies
 41     val = X(i);
 42     if val == 0
 43         alp(i) = 0;
 44     else
 45         alp(i) = log(val)/log(N);
 46     end
 47 end
 48
 49 bet = zeros(numReactions,1);
 50 for i = 1:numReactions
 51     val = cValues(i);
 52     if val ~= 0
 53         bet(i) = log(val)/log(N);
 54     end
 55 end
 56
```

```matlab
 57 V_pos = -V;
 58 V_pos(V_pos < 0) = 0;
 59
 60 % get gamma value from largest of candidates
 61 gam = -Inf;
 62 for i = 1:numSpecies
 63     for k = 1:numReactions
 64         if V(i,k) ~= 0
 65             gam_can = bet(k) + dot(V_pos(:,k),alp) - alp(i);
 66             if gam_can > gam
 67                 gam = gam_can;
 68             end
 69         end
 70     end
 71 end
 72
 73 % get rho value from largest of candidates
 74 rho = Inf;
 75 for k = 1:numReactions
 76     for i = 1:numSpecies
 77         if V(i,k) ~= 0
 78             rho_can = alp(i);
 79             if rho_can < rho
 80                 rho = rho_can;
 81             end
 82         end
 83     end
 84 end
 85
 86 L = ceil(abs(log(err)));
 87
 88 % should have three levels
 89 if L <= 2
 90     l_0 = 0;
 91 else
 92     l_0 = L - 2;
 93 end
 94
 95 num_levels = L - l_0 + 1;
 96
 97 % get required number of coarsest trajectories
 98 n_0 = 4 * ceil( (N^-rho * N^-gam * err^-2) / 4 );
 99
100 % get level step sizes and required number of trajectories
101 h_l = zeros(num_levels, 1);
102 n_l = zeros(num_levels, 1);
103 for i = 1:num_levels
104     l = l_0 + i - 1;
105     h_l(i) = tfinal/(M^l);
106     n_l(i) = ceil( N^-rho * N^gam * (L - l_0) * h_l(i) * err^-2 );
107 end
108
109 % make each n_l divisible by 4
110 for i = 1:num_levels
111     val = n_l(i);
112     while mod(val,4) ~= 0
113         val = val+1;
114     end
```

```matlab
115      n_l(i) = val;
116 end
117
118 % print information if required
119 if verbose_flag
120     fprintf('N:\t%d\n', N);
121     fprintf('Gamma:\t%d\n', gam);
122     fprintf('Rho:\t%d\n', rho);
123     fprintf('M:\t%d\n', M);
124     fprintf('Error:\t%d\n', err);
125     fprintf('Granularities:\n\n');
126         disp(h_l);
127     fprintf('Number of trajectories at each level:\n\n');
128         disp(n_0);
129         disp(n_l(2:num_levels));
130 end
131
132 num_trajectories = sum(n_l);
133
134 interval = tfinal/(numSteps+1);
135
136 % level 0
137 [~, Mean_coarse, ~] = SSAGen_parfor_tauleap(SysInf, tfinal, 0, 0, h_l(1), speciesToGraph,↵
numSteps+1, 0, n_0);
138
139 Y = Mean_coarse;
140 time = linspace(0,tfinal,numSteps+1);
141
142 % ---------------------------- % start MLMC
143
144 for i = 2:num_levels
145
146     num_runs = n_l(i);
147
148     Y_sub = zeros( num_runs , numSpecies, numSteps+1);
149
150     parfor k = 1:num_runs
151
152         % setup that level trajectory
153         hl      = h_l(i);
154         hl_1    = M*hl;
155         l           = l_0 + i - 1;
156         zl          = X;
157         zl_1        = X;
158         Zl          = zeros(numSpecies, numSteps+1);
159         Zl_1        = zeros(numSpecies, numSteps+1);
160         Zl(:,1)     = X;
161         Zl_1(:,1)   = X;
162         t           = 0;
163         n           = 2;
164
165         while n <= (numSteps+1)
166
167             lam_bot = calculatePropensities(zl_1)';
168             A = zeros(numReactions, 3);
169
170             for j = 1:M
171
```

```matlab
172                  lam_top = calculatePropensities(zl)';
173
174                  % (a)
175                  A(:,1) = min(lam_top, lam_bot);
176                  A(:,2) = lam_top - A(:,1);
177                  A(:,3) = lam_bot - A(:,1);
178
179                  % (b)
180                  Lam = poissrnd(A*hl);
181
182                  % (c)
183                  delta_top = Lam(:,1) + Lam(:,2);
184                  delta_bot = Lam(:,1) + Lam(:,3);
185                  zl = zl + V*delta_top;
186                  zl_1 = zl_1 + V*delta_bot;
187
188                  zl(speConstIndecies) = speValues(speConstIndecies);
189                  zl_1(speConstIndecies) = speValues(speConstIndecies);
190                  zl = calculateSpecifiedTotals(zl);
191                  zl_1 = calculateSpecifiedTotals(zl_1);
192
193              end
194
195              t = t + hl_1;
196
197              % record
198              if t > ((n-1)*interval)
199                  Zl(:,n) = zl;
200                  Zl_1(:,n) = zl_1;
201                  n = n + 1;
202              end
203          end
204
205          data = Zl - Zl_1;
206          Y_sub(k,:,:) = data;
207
208      end
209
210      Mean_level = zeros(numSpecies, numSteps+1);
211
212      for i = 1:numSpecies
213          for j = 1:(numSteps+1)
214              Mean(i,j) = mean( Y_sub(:,i,j) );
215          end
216      end
217
218      Y = Y + Mean_level;
219
220 end
221
222 Step = h_l(length(h_l));
223
224 end
```

```matlab
1 function [time, Y] = RREGen(SysInf, tfinal, verbose_flag, stiff_flag)
2
3 numSpecies          = SysInf.numSpecies;
4 numReactions        = SysInf.numReactions;
5 speNames            = SysInf.speNames;
6 speValues           = SysInf.speValues;
7 cNames              = SysInf.cNames;
8 cValues             = SysInf.cValues;
9 speConstIndecies    = SysInf.speConstIndecies;
10 totalsIndecies      = SysInf.totalsIndecies;
11 VHolder             = SysInf.VHolder;
12
13 if verbose_flag
14     disp( sprintf('\nNumber of species types:\n') ); disp(numSpecies);
15     disp( sprintf('\nNumber of reactions:\n') ); disp(numReactions);
16 end
17
18 if verbose_flag
19     disp( sprintf('\nParameter names:\n') ); disp(cNames);
20     disp( sprintf('\nParameter values:\n') ); disp(cValues);
21 end
22
23 if verbose_flag
24     disp( sprintf('\nSpecies'' names:\n') ); disp(speNames);
25     disp( sprintf('\nSpecies'' initial amounts:\n') ); disp(speValues);
26 end
27
28 % holder strings for RHS of RREs
29 for i = 1:numReactions
30     A{i} = '';
31 end
32 A = A';
33
34 V = VHolder.V;
35 vNumOfReactant = VHolder.vNumOfReactant;
36 vReactant = VHolder.vReactant;
37 vDimerMap = VHolder.vDimerMap;
38
39 if verbose_flag
40     disp( sprintf('Stoichiometric Matrix:\n') ); disp(V);
41 end
42
43 for i = 1:numReactions
44
45     % set initially to that reaction's parameter
46     format long;
47     A{i} = strcat( A{i} , num2str( cValues(i) ) );
48
49     % multiply current value by each reactant's value if applicable, and account for↵
dimerisation reactions
50     for k = 1:(vNumOfReactant(i));
51
52         curSpeciesIndex = vReactant(i,k);
53
54         A{i} = strcat( A{i} , sprintf('*X(%d)',curSpeciesIndex) );
55
56         % determine if reactant is part of a dimerisation reaction using dimerisation map,↵
then alter propensity accordingly
```

```matlab
57              dimer_number = vDimerMap(curSpeciesIndex, i);
58              if dimer_number > 1
59                  for j = 1:(dimer_number-1)
60                      A{i} = strcat( A{i} , sprintf('*(X(%d)-%d)', curSpeciesIndex, j ) );
61                  end
62
63                  A{i} = strcat( A{i} , sprintf('/%d', factorial(dimer_number) ) );
64              end
65          end
66 end
67
68 fid = fopen('RRE_functions.m','w');
69 fprintf(fid,'function dXdt = RRE_functions(t,X)');
70
71 fprintf(fid,'\n\ndXdt = zeros(%d,1);\n\n', numSpecies);
72
73 for i = 1:numSpecies
74
75      fprintf(fid,'dXdt(%d) = 0',i);
76
77      if ~ismember(i,speConstIndecies)
78          for j = 1:numReactions
79              if V(i,j) ~= 0
80                  fprintf(fid, ' + %d*%s', V(i,j), A{j});
81              end
82          end
83      end
84
85      fprintf(fid,' ;\n');
86
87 end
88
89 fprintf(fid,'\nend');
90 fclose(fid);
91
92 %tspan = linspace(0,tfinal,tfinal);
93 tspan = [0 tfinal];
94
95 disp( sprintf('==========Starting Solver==========') )
96
97 if stiff_flag
98      [time,y] = ode15s(@RRE_functions,tspan,speValues);
99 else
100     [time,y] = ode45(@RRE_functions,tspan,speValues);
101 end
102
103 Y = y';
104
105 if length(totalsIndecies) ~= 0
106     for i = 1:length(time)
107         Y(:,i) = calculateSpecifiedTotals(Y(:,i));
108     end
109 end
110
111 if verbose_flag
112     disp(sprintf('\nSpecies'' final amounts:\n'));
113     Amount = Y(:,length(Y));
114     dataTable = table(Amount,'RowNames',speNames);
```

```
115      disp(dataTable);
116 end
117
118 end
```

```matlab
 1 function [Y, X, time, run_time] = SingleTrajectory(V, X, speConstIndecies, numSpecies, ↵
speValues, tfinal, recordStep, verbose_flag)
 2
 3 % set max muber of data points for each chunk of the recorded values matrix
 4 numMaxDataPoints = 10000;
 5
 6 Y = zeros(numSpecies, numMaxDataPoints);
 7 time = zeros(1, numMaxDataPoints);
 8
 9 % assign initial values recorded values
10 Y( : , 1 ) = X;
11 time(1) = 0;
12
13 % initial values
14 t = 0;
15 count = 1;
16
17 speConstIndeciesLength = length(speConstIndecies);
18
19 if verbose_flag
20     disp( sprintf('\n=================STARTING SSA=============\n') );
21 end
22
23 tic
24
25 while t < tfinal
26
27     %------------------------------------------------------------------
28     % calculate value of each propensity at that step
29     A = calculatePropensities(X);
30     %------------------------------------------------------------------
31
32     asum = sum(A);
33
34     % break out of simulation if all species are consumed
35     if asum == 0
36         if verbose_flag
37             disp(sprintf('\n=========REACTION HALTED - ALL SPECIES COMSUMED=======\n'));
38             disp(sprintf('Final time was %f', t) );
39         end
40
41         Y( : , ceil(count/recordStep) + 1 ) = X;
42         time( ceil(count/recordStep) + 1 ) = t;
43
44         break
45     end
46
47     j = min( find( rand < cumsum(A/asum) ) );
48     tau = log(1/rand)/asum;
49
50     X = X + V(:,j);
51
52     %------------------------------------------------------------------
53     X = calculateSpecifiedTotals(X);
54     %------------------------------------------------------------------
55
56     for i = 1:speConstIndeciesLength
57         X(speConstIndecies(i)) = speValues(speConstIndecies(i));
```

```matlab
58         end
59
60         t = t + tau;
61
62         if mod(count, recordStep) == 0
63             Y( : , (count/recordStep) + 1) = X;
64             time(count/recordStep + 1) = t;
65         end
66
67         count = count + 1;
68     end
69
70 run_time = toc;
71
72 Y = Y( : ,1:(floor(count/recordStep)) );
73 time = time( 1:(floor(count/recordStep)) );
74
75 if verbose_flag
76     disp( sprintf('\n%d steps taken\n', count ) );
77     disp( sprintf('%d steps recorded\n', floor(count/recordStep) ) );
78 end
79
80 end
```

```matlab
 1 function [Y, X, time, run_time] = SingleTrajectory_cle(V, X, speConstIndecies, numSpecies,↙
speValues, tfinal, recordStep, verbose_flag, tau)
 2
 3 % set max muber of data points for each chunk of the recorded values matrix
 4 numMaxDataPoints = 10000;
 5
 6 Y = zeros(numSpecies, numMaxDataPoints);
 7 time = zeros(1, numMaxDataPoints);
 8
 9 % assign initial values recorded values
10 Y( : , 1 ) = X;
11 time(1) = 0;
12
13 % initial values
14 t = 0;
15 count = 1;
16
17 speConstIndeciesLength = length(speConstIndecies);
18
19 if verbose_flag
20     disp( sprintf('\n================STARTING SSA WITH CLE=============\n') );
21 end
22
23 tic
24
25 while t < tfinal
26
27     %----------------------------------------------------------------
28     % calculate value of each propensity at that step
29     A = calculatePropensities(X);
30     %----------------------------------------------------------------
31
32     % break out of simulation if all species are consumed
33     if cumsum(A) == 0
34         if verbose_flag
35             disp(sprintf('\n========REACTION HALTED - ALL SPECIES COMSUMED=======\n'));
36             disp(sprintf('Final time was %f', t) );
37         end
38
39         Y( : , floor(count/recordStep) + 1) = X;
40         time( floor(count/recordStep) + 1) = t;
41
42         break
43     end
44
45     % get sampling of random variables and sub into CLE formula
46     d = tau*A + sqrt( abs(tau*A) ) * rand;
47
48     % update values
49     X = X + V * d';
50
51     %----------------------------------------------------------------
52     X = calculateSpecifiedTotals(X);
53     %----------------------------------------------------------------
54
55     for i = 1:speConstIndeciesLength
56         X(speConstIndecies(i)) = speValues(speConstIndecies(i));
57     end
```

```matlab
58
59      t = t + tau;
60
61      if mod(count, recordStep) == 0
62          Y( : , (count/recordStep) + 1) = X;
63          time(count/recordStep + 1) = t;
64      end
65
66      count = count + 1;
67 end
68
69 run_time = toc;
70
71 Y = Y( : ,1:(floor(count/recordStep)) );
72 time = time( 1:(floor(count/recordStep)) );
73
74 if verbose_flag
75     disp( sprintf('\n%d steps taken\n', count-1 ) );
76     disp( sprintf('%d steps recorded\n', floor(count/recordStep)-1 ) );
77 end
78
79 end
```

```matlab
 1 function [Y, X, time, run_time] = SingleTrajectory_tauleap(V, X, speConstIndecies, numSpecies,↙
   speValues, tfinal, recordStep, verbose_flag, tau)
 2
 3 % set max muber of data points for each chunk of the recorded values matrix
 4 numMaxDataPoints = 10000;
 5
 6 Y = zeros(numSpecies, numMaxDataPoints);
 7 time = zeros(1, numMaxDataPoints);
 8
 9 % assign initial values recorded values
10 Y( : , 1 ) = X;
11 time(1) = 0;
12
13 % initial values
14 t = 0;
15 count = 1;
16
17 speConstIndeciesLength = length(speConstIndecies);
18
19 if verbose_flag
20     disp( sprintf('\n=================STARTING SSA WITH TAU_LEAPING=============\n') );
21 end
22
23 tic
24
25 while t < tfinal
26
27     %----------------------------------------------------------------
28     % calculate value of each propensity at that step
29     A = calculatePropensities(X);
30     %----------------------------------------------------------------
31
32     % break out of simulation if all species are consumed
33     if cumsum(A) == 0
34         if verbose_flag
35             disp(sprintf('\n=========REACTION HALTED - ALL SPECIES COMSUMED=======\n'));
36             disp(sprintf('Final time was %f', t) );
37         end
38
39         Y( : , floor(count/recordStep) + 1) = X;
40         time( floor(count/recordStep) + 1) = t;
41
42         break
43     end
44
45     % get sampling of poisson random variables
46     pois_rand_vars = poissrnd(A*tau);
47
48     % update values
49     X = X + V * pois_rand_vars';
50
51     %----------------------------------------------------------------
52     X = calculateSpecifiedTotals(X);
53     %----------------------------------------------------------------
54
55     for i = 1:speConstIndeciesLength
56         X(speConstIndecies(i)) = speValues(speConstIndecies(i));
57     end
```

```matlab
58
59      t = t + tau;
60
61      if mod(count, recordStep) == 0
62          Y( : , (count/recordStep) + 1) = X;
63          time(count/recordStep + 1) = t;
64      end
65
66      count = count + 1;
67 end
68
69 run_time = toc;
70
71 Y = Y( : ,1:(floor(count/recordStep)) );
72 time = time( 1:(floor(count/recordStep)) );
73
74 if verbose_flag
75      disp( sprintf('\n%d steps taken\n', count-1 ) );
76      disp( sprintf('%d steps recorded\n', floor(count/recordStep)-1 ) );
77 end
78
79 end
```

```matlab
1  function Y = SSA_gpu(filename, SysInf, tfinal, verbose_flag)
2
3  numSpecies          = SysInf.numSpecies;
4  numReactions        = SysInf.numReactions;
5  speNames            = SysInf.speNames;
6  speValues           = SysInf.speValues;
7  cNames              = SysInf.cNames;
8  cValues             = SysInf.cValues;
9  speConstIndecies    = SysInf.speConstIndecies;
10 totalsIndecies      = SysInf.totalsIndecies;
11 VHolder             = SysInf.VHolder;
12 gpu = gpuDevice();
13
14 if verbose_flag
15     disp( sprintf('GPU Device detected:\n') );
16     disp( gpu );
17 end
18
19 V = VHolder.V;
20 vNumOfReactant = VHolder.vNumOfReactant;
21 vReactant = VHolder.vReactant;
22 vDimerMap = VHolder.vDimerMap;
23
24 fid = fopen('fireGpuTrajectories.m','w');
25 fprintf(fid,'function Y = fireGpuTrajectories(VHolder, verbose_flag)\n\n');
26
27 fprintf(fid, 'V = VHolder.V;\n\n');
28
29 for i = 1:numSpecies
30     fprintf(fid, 'x%d = %d;\n', i, speValues(i));
31 end
32
33 fprintf(fid, '\ntfinal = %d;\n\n', tfinal);
34
35 fprintf(fid, '\tfunction [input');
36 for i = 1:numSpecies
37     fprintf(fid, ', x%d', i);
38 end
39 fprintf(fid, ']');
40
41 fprintf(fid, ' = fire_single_gpu_trajectory(input');
42 for i = 1:numSpecies
43     fprintf(fid, ', x%d', i);
44 end
45 fprintf(fid, ')\n\n');
46
47 format long
48
49 for i = 1:numReactions
50     fprintf(fid, '\t\tc%d = %e;\n', i, cValues(i));
51 end
52
53 fprintf(fid, '\n\t\tt = 0;\n');
54
55 fprintf(fid, '\n\t\twhile t < tfinal\n\n');
56
57 for i = 1:numReactions
58
```

```matlab
59      % set initially to that reaction's parameter
60      format long;
61      fprintf(fid, '\t\t\ta%d = (%e)', i, cValues(i) );
62
63      % multiply current value by each ractant's value if applicable, and account for↙
   dimerisation reactions
64      for k = 1:(vNumOfReactant(i));
65
66          curSpeciesIndex = vReactant(i,k);
67
68          fprintf(fid,'*x%d', curSpeciesIndex);
69
70          % determine is reactant is part of a dimerisation reaction using dimerisation map,↙
   then alter propensity accordingly
71          dimer_number = vDimerMap(curSpeciesIndex, i);
72          if dimer_number > 1
73              count = 1;
74              while (count < dimer_number)
75                  fprintf(fid,'*(x%d-%d)', curSpeciesIndex, count);
76                  count = count + 1;
77              end
78
79              fprintf(fid,'/%d', factorial(dimer_number) );
80          end
81      end
82
83      fprintf(fid, ';\n', i, cValues(i) );
84 end
85
86 fprintf(fid, '\n\t\t\tasum =');
87 for i = 1:numReactions
88      fprintf(fid, ' + a%d', i);
89 end
90 fprintf(fid, ';\n\n');
91
92 fprintf(fid, '\t\t\tif asum == 0\n\t\t\t\treturn\n\t\t\tend\n\n');
93
94 for i = 1:numReactions
95      fprintf(fid, '\t\t\ta_tot_%d = (', i);
96      for j = 1:i
97          fprintf(fid,'+a%d', j);
98      end
99      fprintf(fid, ')/asum;\n');
100 end
101
102 fprintf(fid, '\n\t\t\tj = 1;\n\n');
103 fprintf(fid, '\t\t\trand_num = rand;\n\n');
104
105 fprintf(fid, '\t\t\tif a_tot_1 > rand_num\n');
106 fprintf(fid, '\t\t\t\tj = 1;\n');
107
108 for i = 2:numReactions
109      fprintf(fid, '\t\t\telseif a_tot_%d > rand_num\n', i);
110      fprintf(fid, '\t\t\t\tj = %d;\n', i);
111 end
112
113 fprintf(fid, '\t\tend\n\n');
114
```

```matlab
115 fprintf(fid, '\t\t\ttau = log(1/rand)/asum;\n\n');
116
117 for i = 1:numSpecies
118     if ~ismember(i,speConstIndecies)
119         fprintf(fid, '\t\t\tx%d = x%d + V(%d,j);\n', i, i, i);
120     end
121 end
122
123 fprintf(fid, '\n');
124 SBMLModel = TranslateSBML(filename);
125 count = 0;
126 for i = 1:length(SBMLModel.rule)
127     curRule = SBMLModel.rule(i);
128
129     for j = 1:length( speNames )
130         if strcmp( speNames(j), curRule.variable )
131
132             totalsIndecies(count+1) = j;
133             count = count + 1;
134             fprintf(fid, '\t\t\tx%d =', j);
135
136             % token string array
137             ruleToks = strsplit( curRule.formula ,'+');
138
139             for l = 1:length( ruleToks )
140                 for m = 1:length(speNames)
141                     if strcmp( speNames(m), ruleToks(l) )
142                         fprintf(fid, ' + x%d', m);
143                     end
144                 end
145             end
146
147             fprintf(fid, ';\n', m);
148
149             break;
150         end
151     end
152 end
153
154 fprintf(fid, '\n\t\t\tt = t + tau;\n\n');
155
156 fprintf(fid, '\t\tend\n\n');
157 fprintf(fid, '\tend');
158
159 fprintf(fid, '\n\nnum_trajectories = 10000;\n');
160
161 fprintf(fid, 'trial_nums = linspace(1,num_trajectories, num_trajectories)'';\n' );
162 fprintf(fid, 'inputs = gpuArray(trial_nums);\n' );
163
164 fprintf(fid, '\n[g_trial');
165 for i = 1:numSpecies
166     fprintf(fid, ', g_x%d', i);
167 end
168 fprintf(fid, '] = arrayfun(@fire_single_gpu_trajectory, inputs');
169 for i = 1:numSpecies
170     fprintf(fid, ', x%d', i);
171 end
172 fprintf(fid, ');\n\n');
```

```matlab
173
174 fprintf(fid, 'trials = gather(g_trial);\n');
175
176 fprintf(fid, 'Y = [');
177 for i = 1:numSpecies
178     fprintf(fid, ' gather(g_x%d)', i);
179 end
180 fprintf(fid, '];\n\n');
181
182 fprintf(fid,'\nend');
183
184 fclose(fid);
185
186 Y = fireGpuTrajectories(VHolder, verbose_flag);
187 wait(gpu);
188
189 Mean = zeros(numSpecies, 1);
190 Std_dev = zeros(numSpecies, 1);
191 for i = 1:numSpecies
192     data = Y( : , i );
193     Mean(i) = mean( data );
194     Std_dev(i) = std( data );
195 end
196
197 if verbose_flag
198     dataTableMean = table(Mean,'RowNames',speNames);
199     disp(dataTableMean);
200     dataTableStddev = table(Std_dev,'RowNames',speNames);
201     disp(dataTableStddev);
202 end
203
204 end
```

```matlab
1 function SysInf = SSA_setup(filename, verbose_flag)
2
3 if verbose_flag
4     disp(' ');
5 end
6
7 arg_type = class(filename);
8
9 switch arg_type
10     case 'char'
11         SBMLModel = TranslateSBML(filename);
12     case 'struct'
13         SBMLModel = filename;
14 end
15
16
17 % determine number of reactions and species present in the model
18 numReactions = length(SBMLModel.reaction);
19 numSpecies = length(SBMLModel.species);
20
21 if verbose_flag
22     disp( sprintf('\nNumber of species types:\n') ); disp(numSpecies);
23     disp( sprintf('\nNumber of reactions:\n') ); disp(numReactions);
24 end
25
26 [cNames, cValues] = GetParameters(SBMLModel);
27
28 if verbose_flag
29     [cNames_us, cValues_us] = GetAllParameters(SBMLModel);
30     cNames_us = cNames_us';
31     cValues_us = cValues_us';
32
33     disp(sprintf('\nParameter Values:\n'))
34     Value = cValues_us;
35     dataTable = table(Value,'RowNames',cNames_us);
36     disp(dataTable);
37 end
38
39 % get species names and values, set X to initial values
40 [speNames, speValues] = GetSpecies(SBMLModel);
41 speNames = speNames';
42 speValues = speValues';
43
44 if verbose_flag
45     disp(sprintf('\nSpecies'' initial amounts:\n'))
46     Amount = speValues;
47     dataTable = table(Amount,'RowNames',speNames);
48     disp(dataTable);
49 end
50
51 % matricies to hold propensity values, number of species present after each step, and the↙
length of each step
52 A = zeros(numReactions,1);
53
54 VHolder = StoichiometricMatricesHolder(SBMLModel);
55
56 % will generate 'calculatePropensities.m' file
57 GeneratePropensityCalculatorFile(SBMLModel, VHolder);
```

```matlab
58
59 % will generate 'calculateSpecifiedTotals.m' file
60 totalsIndecies = GenerateSpecifiedTotalsCalculatorFile(SBMLModel);
61
62 % determine if any species have a boundary condition and get their indecies
63 speConstIndecies = GetConstantSpeciesIndecies(SBMLModel);
64
65 SysInf = SystemInformationHolder;
66
67 SysInf.numSpecies        = numSpecies;
68 SysInf.numReactions      = numReactions;
69 SysInf.speNames          = speNames;
70 SysInf.speValues         = speValues;
71 SysInf.cNames            = cNames;
72 SysInf.cValues           = cValues;
73 SysInf.speConstIndecies  = speConstIndecies;
74 SysInf.totalsIndecies    = totalsIndecies;
75 SysInf.VHolder           = VHolder;
76
77 end
```

```matlab
 1 function [time, Y] = SSAGen(SysInf, tfinal, recordStep, verbose_flag, split_flag)
 2
 3 numSpecies          = SysInf.numSpecies;
 4 numReactions        = SysInf.numReactions;
 5 speNames            = SysInf.speNames;
 6 speValues           = SysInf.speValues;
 7 cNames              = SysInf.cNames;
 8 cValues             = SysInf.cValues;
 9 speConstIndecies    = SysInf.speConstIndecies;
10 totalsIndecies      = SysInf.totalsIndecies;
11 VHolder             = SysInf.VHolder;
12
13 % initial values
14 X = speValues;
15
16 % extract V from VHolder and display
17 V = VHolder.V;
18
19 if verbose_flag
20     disp( sprintf('Stoichiometric Matrix:\n') ); disp(V);
21 end
22
23 % matricies to hold propensity values, number of species present after each step, and the↙
length of each step
24 A = zeros(numReactions,1);
25
26 % Actually do SSA ------------- %
27 [Y, X, time, run_time] = SingleTrajectory(V, X, speConstIndecies, numSpecies, speValues,↙
tfinal, recordStep, verbose_flag);
28 % ---------------------------- %
29
30 if verbose_flag
31     disp(sprintf('\nSpecies'' final amounts:\n'));
32     Amount = X;
33     dataTable = table(Amount,'RowNames',speNames);
34     disp(dataTable);
35 end
36
37 if verbose_flag
38     disp(' ');
39 end
40
41 end
```

```matlab
1 function [time, Y] = SSAGen(SysInf, tfinal, recordStep, verbose_flag, tau)
2
3 numSpecies          = SysInf.numSpecies;
4 numReactions        = SysInf.numReactions;
5 speNames            = SysInf.speNames;
6 speValues           = SysInf.speValues;
7 cNames              = SysInf.cNames;
8 cValues             = SysInf.cValues;
9 speConstIndecies    = SysInf.speConstIndecies;
10 totalsIndecies      = SysInf.totalsIndecies;
11 VHolder             = SysInf.VHolder;
12
13 % initial values
14 X = speValues;
15
16 % extract V from VHolder and display
17 V = VHolder.V;
18 if verbose_flag
19     disp( sprintf('Stoichiometric Matrix:\n') ); disp(V);
20 end
21
22 % attempt to pick tau if not specified - *extremely* crude estimate
23 if tau == 0
24     % Single SSA trajectory to help determine good tau
25     [Y, X, time, run_time] = SingleTrajectory(V, X, speConstIndecies, numSpecies, speValues,↙
tfinal, recordStep, verbose_flag);
26     tau = ( time(length(time)) / ( length(time)*recordStep ) ) * 3 ;
27 end
28
29 if verbose_flag
30     disp( sprintf('Chosen value for tau:\n') ); disp(tau);
31 end
32
33 X = speValues;
34
35 tic
36 [Y, X, time, run_time] = SingleTrajectory_cle(V, X, speConstIndecies, numSpecies, speValues,↙
tfinal, recordStep, verbose_flag, tau);
37 time_with_leap = toc;
38
39 if verbose_flag
40     disp(sprintf('\nSpecies'' final amounts:\n'));
41     Amount = X;
42     dataTable = table(Amount,'RowNames',speNames);
43     disp(dataTable);
44 end
45
46 if verbose_flag
47     disp(' ');
48 end
49
50 end
```

```matlab
 1 function varargout = SSAGen_parfor(SysInf, tfinal, recordStep, verbose_flag, speciesToGraph,↵
numSteps, graph_flag)
 2
 3 numSpecies          = SysInf.numSpecies;
 4 numReactions        = SysInf.numReactions;
 5 speNames            = SysInf.speNames;
 6 speValues           = SysInf.speValues;
 7 cNames              = SysInf.cNames;
 8 cValues             = SysInf.cValues;
 9 speConstIndecies    = SysInf.speConstIndecies;
10 totalsIndecies      = SysInf.totalsIndecies;
11 VHolder             = SysInf.VHolder;
12
13 if verbose_flag
14     disp(' ');
15 end
16
17 % set max muber of data points for each chunk of the recorded values matrix
18 numMaxDataPoints = 10008;
19
20 % set X to initial values
21 X = speValues;
22
23 % matrix to hold propensity values, number of constant species
24 speConstIndeciesLength = length(speConstIndecies);
25 A = zeros(numReactions,1);
26
27 % extract V from VHolder and display
28 V = VHolder.V;
29 if verbose_flag
30     disp( sprintf('\nStoichiometric Matrix:\n') ); disp(V);
31 end
32
33 num_cores = feature('numCores');
34
35 if verbose_flag
36     disp( sprintf('\nDetected %d CPU cores\n', num_cores) );
37 end
38
39 %Y = zeros(numMaxDataPoints, numSpecies);
40 Y = zeros(numSpecies, numSteps, numMaxDataPoints);
41 %time = zeros(1, numMaxDataPoints);
42
43 % begin SSA algorithm
44
45 if verbose_flag
46     disp( sprintf('\n============== STARTING SSA FOR HISTOGRAM =============='  ) );
47     disp( sprintf(  'Remember - this part takes a while. Please be patient.\n') );
48 end
49
50 halt_flag = 0;
51 time = linspace(0,tfinal,numSteps);
52 interval  = tfinal/numSteps;
53
54 parfor l = 1:numMaxDataPoints
55
56     % initial values
57     t = 0;
```

```matlab
58        n = 2;
59        X = speValues;
60        A = zeros(numReactions,1);
61        Y_sub = zeros(numSpecies, numSteps);
62        Y_sub(:,1) = X;
63
64        while n <= numSteps
65
66            next_step = (n-1)*interval;
67
68            %----------------------------------------------------------------
69            % calculate value of each propensity at that step
70            A = calculatePropensities(X);
71            %----------------------------------------------------------------
72
73            asum = sum(A);
74
75            % break out of simulation if all species are consumed
76            if asum == 0
77                halt_flag = 1;
78                break
79            end
80
81            j = find( rand < cumsum(A/asum) , 1 );
82            tau = log(1/rand)/asum;
83
84            X = X + V(:,j);
85
86            %----------------------------------------------------------------
87            X = calculateSpecifiedTotals(X);
88            %----------------------------------------------------------------
89
90            for i = 1:speConstIndeciesLength
91                X(speConstIndecies(i)) = speValues(speConstIndecies(i));
92            end
93
94            t = t + tau;
95
96            if t > next_step
97                Y_sub(:,n) = X';
98                n = n + 1;
99            end
100
101       end
102
103       Y(:,:,l) = Y_sub;
104
105 end
106
107 if halt_flag && verbose_flag
108     disp(sprintf('\n=========REACTION HALTED - ALL SPECIES COMSUMED=======\n'));
109 end
110
111 if verbose_flag
112     disp(' ');
113 end
114
115 Y_last = zeros(numMaxDataPoints, numSpecies);
```

```matlab
116
117 % get means, standard deviations, last slice data
118 Mean    = zeros(numSpecies, numSteps);
119 Std     = zeros(numSpecies, numSteps);
120 for i = 1:numSpecies
121     for j = 1:numSteps
122         data = Y(i,j,:);
123         if j == numSteps
124             Y_last(:,i) = data;
125         end
126         Mean(i,j) = mean(data);
127         Std(i,j) = std(data);
128     end
129 end
130
131 specific_species_flag = 0;
132
133 if ~isempty(speciesToGraph)
134     stop_point = length(speciesToGraph);
135     specific_species_flag  = 1;
136 else
137     stop_point = numSpecies;
138 end
139
140 warning('off','MATLAB:legend:IgnoringExtraEntries');
141
142 if graph_flag
143     for i = 1:stop_point
144         if specific_species_flag
145             index = speciesToGraph(i);
146         else
147             index = i;
148         end
149
150         data = zeros(numSteps, numMaxDataPoints);
151         data(1:numSteps,1:numMaxDataPoints) = Y(index,:,:);
152         data = data';
153
154         figure;
155
156         % graph boxplot for each slice
157         subplot(2,1,1);
158         boxplot( data , time, 'plotstyle', 'compact');
159         legend( findobj(gca,'Tag','Box'),speNames(index) );
160         xlabel('Time','FontSize',12, 'FontName', 'Helvetica');
161         ylabel('Number of Species','FontSize',12,'FontName', 'Helvetica');
162         title('Box and whisker plot of species vs time at given intervals','FontSize',↵
16,'FontName', 'Helvetica');
163
164         % graph means +- standard deviations for each slice
165         subplot(2,1,2);
166         hold all
167         plot( time, Mean(index,:), 'b-o');
168         plot( time, Mean(index,:) - Std(index,:), 'c');
169         plot( time, Mean(index,:) + Std(index,:), 'c');
170         legend( findobj(gca,'Tag','Box'),speNames(index) );
171         xlabel('Time','FontSize',12, 'FontName', 'Helvetica');
172         ylabel('Number of Species','FontSize',12,'FontName', 'Helvetica');
```

```matlab
173          title('Box and whisker plot of species vs time at given intervals','FontSize',↵
16,'FontName', 'Helvetica');
174      end
175 end
176
177 Mean_last = Mean(:,numSteps);
178 Std_dev_last = Std(:,numSteps);
179
180 varargout{1} = Y_last;
181 varargout{2} = Mean;
182 varargout{3} = Std;
183
184 if verbose_flag
185     dataTableMean = table(Mean_last,'RowNames',speNames);
186     disp(dataTableMean);
187     dataTableStddev = table(Std_dev_last,'RowNames',speNames);
188     disp(dataTableStddev);
189 end
190
191 end
```

```matlab
  1 function varargout = SSAGen_parfor_cle(SysInf, tfinal, recordStep, verbose_flag, tau,↵
speciesToGraph, numSteps, graph_flag)
  2
  3 numSpecies          = SysInf.numSpecies;
  4 numReactions        = SysInf.numReactions;
  5 speNames            = SysInf.speNames;
  6 speValues           = SysInf.speValues;
  7 cNames              = SysInf.cNames;
  8 cValues             = SysInf.cValues;
  9 speConstIndecies    = SysInf.speConstIndecies;
 10 totalsIndecies      = SysInf.totalsIndecies;
 11 VHolder             = SysInf.VHolder;
 12
 13 if verbose_flag
 14     disp(' ');
 15 end
 16
 17 % set max muber of data points for each chunk of the recorded values matrix
 18 numMaxDataPoints = 10008;
 19
 20 % set X to initial values
 21 X = speValues;
 22
 23 % matrix to hold propensity values, number of constant species
 24 speConstIndeciesLength = length(speConstIndecies);
 25 A = zeros(numReactions,1);
 26
 27 % extract V from VHolder and display
 28 V = VHolder.V;
 29 if verbose_flag
 30     disp( sprintf('\nStoichiometric Matrix:\n') ); disp(V);
 31 end
 32
 33 if tau == 0
 34     % Single SSA trajectory to help determine good tau
 35     [Y, X, time, run_time] = SingleTrajectory(V, X, speConstIndecies, numSpecies, speValues,↵
tfinal, recordStep, verbose_flag);
 36     tau = ( time(length(time)) / ( length(time)*recordStep ) ) * 3 ;
 37 end
 38
 39 num_cores = feature('numCores');
 40
 41 if verbose_flag
 42     disp( sprintf('\nDetected %d CPU cores\n', num_cores) );
 43 end
 44
 45 %Y = zeros(numMaxDataPoints, numSpecies);
 46 Y = zeros(numSpecies, numSteps, numMaxDataPoints);
 47 %time = zeros(1, numMaxDataPoints);
 48
 49 % begin SSA with tau-leaping algorithm
 50
 51 if verbose_flag
 52     disp( sprintf('\n============= STARTING parallel SSA with tau-leaping ============='  )↵
);
 53     disp( sprintf(  'Remember - this part takes a while. Please be patient.\n') );
 54 end
 55
```

```matlab
 56 halt_flag = 0;
 57 time = linspace(0,tfinal,numSteps);
 58 interval  = tfinal/numSteps;
 59
 60 parfor l = 1:numMaxDataPoints
 61
 62     % initial values
 63     t = 0;
 64     n = 2;
 65     X = speValues;
 66     A = zeros(numReactions,1);
 67     Y_sub = zeros(numSpecies, numSteps);
 68     Y_sub(:,1) = X;
 69
 70     while n <= numSteps
 71
 72         next_step = (n-1)*interval;
 73
 74         %-----------------------------------------------------------------
 75         % calculate value of each propensity at that step
 76         A = calculatePropensities(X);
 77         %-----------------------------------------------------------------
 78
 79         asum = sum(A);
 80
 81         % break out of simulation if all species are consumed
 82         if asum == 0
 83             halt_flag = 1;
 84             break
 85         end
 86
 87         % get sampling of random variables and sub into CLE formula
 88         d = tau*A + sqrt( abs(tau*A) ) * randn;
 89
 90         % update values
 91         X = X + V * d';
 92
 93         %-----------------------------------------------------------------
 94         X = calculateSpecifiedTotals(X);
 95         %-----------------------------------------------------------------
 96
 97         for i = 1:speConstIndeciesLength
 98             X(speConstIndecies(i)) = speValues(speConstIndecies(i));
 99         end
100
101         t = t + tau;
102
103         if t > next_step
104             Y_sub(:,n) = X';
105             n = n + 1;
106         end
107
108     end
109
110     Y(:,:,l) = Y_sub;
111
112 end
113
```

```matlab
114 if halt_flag && verbose_flag
115     disp(sprintf('\n=========REACTION HALTED - ALL SPECIES COMSUMED=======\n'));
116 end
117
118 if verbose_flag
119     disp(' ');
120 end
121
122 Y_last = zeros(numMaxDataPoints, numSpecies);
123
124 % get means, standard deviations, last slice data
125 Mean    = zeros(numSpecies, numSteps);
126 Std     = zeros(numSpecies, numSteps);
127 for i = 1:numSpecies
128     for j = 1:numSteps
129         data = Y(i,j,:);
130         if j == numSteps
131             Y_last(:,i) = data;
132         end
133         Mean(i,j) = mean(data);
134         Std(i,j) = std(data);
135     end
136 end
137
138 specific_species_flag = 0;
139
140 if ~isempty(speciesToGraph)
141     stop_point = length(speciesToGraph);
142     specific_species_flag  = 1;
143 else
144     stop_point = numSpecies;
145 end
146
147 warning('off','MATLAB:legend:IgnoringExtraEntries');
148
149 if graph_flag
150     for i = 1:stop_point
151         if specific_species_flag
152             index = speciesToGraph(i);
153         else
154             index = i;
155         end
156
157         data = zeros(numSteps, numMaxDataPoints);
158         data(1:numSteps,1:numMaxDataPoints) = Y(index,:,:);
159         data = data';
160
161         figure;
162
163         % graph boxplot for each slice
164         subplot(2,1,1);
165         boxplot( data , time, 'plotstyle', 'compact');
166         legend( findobj(gca,'Tag','Box'),speNames(index) );
167         xlabel('Time','FontSize',12, 'FontName', 'Helvetica');
168         ylabel('Number of Species','FontSize',12,'FontName', 'Helvetica');
169         title('Box and whisker plot of species vs time at given intervals','FontSize',↙
16,'FontName', 'Helvetica');
170
```

```matlab
171          % graph means +- standard deviations for each slice
172          subplot(2,1,2);
173          hold all
174          plot( time, Mean(index,:), 'b-o');
175          plot( time, Mean(index,:) - Std(index,:), 'c');
176          plot( time, Mean(index,:) + Std(index,:), 'c');
177          legend( findobj(gca,'Tag','Box'),speNames(index) );
178          xlabel('Time','FontSize',12, 'FontName', 'Helvetica');
179          ylabel('Number of Species','FontSize',12,'FontName', 'Helvetica');
180        title('Box and whisker plot of species vs time at given intervals','FontSize',↵
16,'FontName', 'Helvetica');
181      end
182 end
183
184 Mean_last = Mean(:,numSteps);
185 Std_dev_last = Std(:,numSteps);
186
187 varargout{1} = Y_last;
188 varargout{2} = Mean;
189 varargout{3} = Std;
190
191 if verbose_flag
192     dataTableMean = table(Mean_last,'RowNames',speNames);
193     disp(dataTableMean);
194     dataTableStddev = table(Std_dev_last,'RowNames',speNames);
195     disp(dataTableStddev);
196 end
197
198 end
```

```matlab
  1 function varargout = SSAGen_parfor_tauleap(SysInf, tfinal, recordStep, verbose_flag, tau,↵
speciesToGraph, numSteps, graph_flag, num_traj)
  2
  3 numSpecies         = SysInf.numSpecies;
  4 numReactions       = SysInf.numReactions;
  5 speNames           = SysInf.speNames;
  6 speValues          = SysInf.speValues;
  7 cNames             = SysInf.cNames;
  8 cValues            = SysInf.cValues;
  9 speConstIndecies   = SysInf.speConstIndecies;
 10 totalsIndecies     = SysInf.totalsIndecies;
 11 VHolder            = SysInf.VHolder;
 12
 13 if verbose_flag
 14     disp(' ');
 15 end
 16
 17 % set max muber of data points for each chunk of the recorded values matrix
 18 if num_traj == 0
 19     numMaxDataPoints = 10008;
 20 else
 21     numMaxDataPoints = num_traj;
 22
 23 % set X to initial values
 24 X = speValues;
 25
 26 % matrix to hold propensity values, number of constant species
 27 speConstIndeciesLength = length(speConstIndecies);
 28 A = zeros(numReactions,1);
 29
 30 % extract V from VHolder and display
 31 V = VHolder.V;
 32 if verbose_flag
 33     disp( sprintf('\nStoichiometric Matrix:\n') ); disp(V);
 34 end
 35
 36 if tau == 0
 37     % Single SSA trajectory to help determine good tau
 38     [Y, X, time, run_time] = SingleTrajectory(V, X, speConstIndecies, numSpecies, speValues,↵
tfinal, recordStep, verbose_flag);
 39     tau = ( time(length(time)) / ( length(time)*recordStep ) ) * 3 ;
 40 end
 41
 42 num_cores = feature('numCores');
 43
 44 if verbose_flag
 45     disp( sprintf('\nDetected %d CPU cores\n', num_cores) );
 46 end
 47
 48 %Y = zeros(numMaxDataPoints, numSpecies);
 49 Y = zeros(numSpecies, numSteps, numMaxDataPoints);
 50 %time = zeros(1, numMaxDataPoints);
 51
 52 % begin SSA with tau-leaping algorithm
 53
 54 if verbose_flag
 55     disp( sprintf('\n============== STARTING parallel SSA with tau-leaping =============='  )↵
);
```

```matlab
56      disp( sprintf(  'Remember - this part takes a while. Please be patient.\n') );
57 end
58
59 halt_flag = 0;
60 time = linspace(0,tfinal,numSteps);
61 interval  = tfinal/numSteps;
62
63 parfor l = 1:numMaxDataPoints
64
65     % initial values
66     t = 0;
67     n = 2;
68     X = speValues;
69     A = zeros(numReactions,1);
70     Y_sub = zeros(numSpecies, numSteps);
71     Y_sub(:,1) = X;
72
73     while n <= numSteps
74
75         next_step = (n-1)*interval;
76
77         %-----------------------------------------------------------------
78         % calculate value of each propensity at that step
79         A = calculatePropensities(X);
80         %-----------------------------------------------------------------
81
82         asum = sum(A);
83
84         % break out of simulation if all species are consumed
85         if asum == 0
86             halt_flag = 1;
87             break
88         end
89
90         % get sampling of poisson random variables
91         pois_rand_vars = poissrnd(A*tau);
92
93         % update values
94         X = X + V * pois_rand_vars';
95
96         %-----------------------------------------------------------------
97         X = calculateSpecifiedTotals(X);
98         %-----------------------------------------------------------------
99
100        for i = 1:speConstIndeciesLength
101            X(speConstIndecies(i)) = speValues(speConstIndecies(i));
102        end
103
104        t = t + tau;
105
106        if t > next_step
107            Y_sub(:,n) = X';
108            n = n + 1;
109        end
110
111    end
112
113    Y(:,:,l) = Y_sub;
```

```matlab
114
115 end
116
117 if halt_flag && verbose_flag
118     disp(sprintf('\n=========REACTION HALTED - ALL SPECIES COMSUMED=======\n'));
119 end
120
121 if verbose_flag
122     disp(' ');
123 end
124
125 Y_last = zeros(numMaxDataPoints, numSpecies);
126
127 % get means, standard deviations, last slice data
128 Mean    = zeros(numSpecies, numSteps);
129 Std     = zeros(numSpecies, numSteps);
130 for i = 1:numSpecies
131     for j = 1:numSteps
132         data = Y(i,j,:);
133         if j == numSteps
134             Y_last(:,i) = data;
135         end
136         Mean(i,j) = mean(data);
137         Std(i,j) = std(data);
138     end
139 end
140
141 specific_species_flag = 0;
142
143 if ~isempty(speciesToGraph)
144     stop_point = length(speciesToGraph);
145     specific_species_flag  = 1;
146 else
147     stop_point = numSpecies;
148 end
149
150 warning('off','MATLAB:legend:IgnoringExtraEntries');
151
152 if graph_flag
153     for i = 1:stop_point
154         if specific_species_flag
155             index = speciesToGraph(i);
156         else
157             index = i;
158         end
159
160         data = zeros(numSteps, numMaxDataPoints);
161         data(1:numSteps,1:numMaxDataPoints) = Y(index,:,:);
162         data = data';
163
164         figure;
165
166         % graph boxplot for each slice
167         subplot(2,1,1);
168         boxplot( data , time, 'plotstyle', 'compact');
169         legend( findobj(gca,'Tag','Box'),speNames(index) );
170         xlabel('Time','FontSize',12, 'FontName', 'Helvetica');
171         ylabel('Number of Species','FontSize',12,'FontName', 'Helvetica');
```

```matlab
172         title('Box and whisker plot of species vs time at given intervals','FontSize',↵
16,'FontName', 'Helvetica');
173
174         % graph means +- standard deviations for each slice
175         subplot(2,1,2);
176         hold all
177         plot( time, Mean(index,:), 'b-o');
178         plot( time, Mean(index,:) - Std(index,:), 'c');
179         plot( time, Mean(index,:) + Std(index,:), 'c');
180         legend( findobj(gca,'Tag','Box'),speNames(index) );
181         xlabel('Time','FontSize',12, 'FontName', 'Helvetica');
182         ylabel('Number of Species','FontSize',12,'FontName', 'Helvetica');
183         title('Box and whisker plot of species vs time at given intervals','FontSize',↵
16,'FontName', 'Helvetica');
184     end
185 end
186
187 Mean_last = Mean(:,numSteps);
188 Std_dev_last = Std(:,numSteps);
189
190 varargout{1} = Y_last;
191 varargout{2} = Mean;
192 varargout{3} = Std;
193
194 if verbose_flag
195     dataTableMean = table(Mean_last,'RowNames',speNames);
196     disp(dataTableMean);
197     dataTableStddev = table(Std_dev_last,'RowNames',speNames);
198     disp(dataTableStddev);
199 end
200
201 end
```

```matlab
 1 function [time, Y] = SSAGen(SysInf, tfinal, recordStep, verbose_flag, tau)
 2
 3 numSpecies         = SysInf.numSpecies;
 4 numReactions       = SysInf.numReactions;
 5 speNames           = SysInf.speNames;
 6 speValues          = SysInf.speValues;
 7 cNames             = SysInf.cNames;
 8 cValues            = SysInf.cValues;
 9 speConstIndecies   = SysInf.speConstIndecies;
10 totalsIndecies     = SysInf.totalsIndecies;
11 VHolder            = SysInf.VHolder;
12
13 % initial values
14 X = speValues;
15
16 % extract V from VHolder and display
17 V = VHolder.V;
18 if verbose_flag
19     disp( sprintf('Stoichiometric Matrix:\n') ); disp(V);
20 end
21
22 % attempt to pick tau if not specified - *extremely* crude estimate
23 if tau == 0
24     % Single SSA trajectory to help determine good tau
25     [Y, X, time, run_time] = SingleTrajectory(V, X, speConstIndecies, numSpecies, speValues,↙
tfinal, recordStep, verbose_flag);
26     tau = ( time(length(time)) / ( length(time)*recordStep ) ) * 3 ;
27 end
28
29 disp( sprintf('Chosen value for tau:\n') ); disp(tau);
30
31 X = speValues;
32
33 tic
34 [Y, X, time, run_time] = SingleTrajectory_tauleap(V, X, speConstIndecies, numSpecies,↙
speValues, tfinal, recordStep, verbose_flag, tau);
35 time_with_leap = toc;
36
37 if verbose_flag
38     disp(sprintf('\nSpecies'' final amounts:\n'));
39     Amount = X;
40     dataTable = table(Amount,'RowNames',speNames);
41     disp(dataTable);
42 end
43
44 if verbose_flag
45     disp(' ');
46 end
47
48 end
```

```matlab
1  classdef StoichiometricMatricesHolder
2
3      properties
4          V;
5          vNumOfReactant;
6          vReactant;
7          vDimerMap;
8      end
9
10     methods
11
12         function object = StoichiometricMatricesHolder(SBMLModel)
13
14             numReactions = length(SBMLModel.reaction);
15             numSpecies = length(SBMLModel.species);
16
17             % get the stoichiometric matrix representing the species changes for each reaction↙
   from the model
18             V = zeros(numSpecies, numReactions);
19
20             % matricies to hold the number of reactants in each reaction, the reactants'↙
   indecies, and the dimer map
21             vNumOfReactant = zeros(numReactions,1);
22             vReactant = zeros(numReactions, numSpecies);
23             vDimerMap = zeros(numSpecies, numReactions);
24
25             maxCount = 0;
26
27             for j = 1:numReactions
28
29                 count = 0;
30                 for i = 1:numSpecies
31
32                     role = DetermineSpeciesRoleInReaction(SBMLModel.species(i), SBMLModel.↙
   reaction(j));
33                     if length(role) > 1
34
35                         V(i,j) = role(1) - role(2);
36
37                         if role(2) > 0
38                             vReactant( j , (count+1) ) = i;
39                             count = count + 1;
40                         end
41
42                         if role(2) >= 2
43                             vDimerMap(i,j) = role(2);
44                         end
45                     else
46                         V(i,j) = 0;
47                     end
48                 end
49
50                 vNumOfReactant(j) = count;
51
52                 if count > maxCount
53                     maxCount = count;
54                 end
55
```

```matlab
56                 end
57
58             object.V = V;
59             object.vNumOfReactant = vNumOfReactant;
60
61             % truncate reactant index matrix to discard unnecessary elements
62             object.vReactant = vReactant( : , 1:(maxCount) );
63
64             % make dimer reactant matrix sparse to discard unnecessary elements
65             object.vDimerMap = vDimerMap;
66
67         end
68
69     end
70
71 end
```

```matlab
 1  classdef SystemInformationHolder
 2
 3      properties
 4          numSpecies;
 5          numReactions;
 6          speNames;
 7          speValues;
 8          cNames;
 9          cValues;
10          speConstIndecies;
11          totalsIndecies;
12          VHolder;
13      end
14
15  end
```