Simplifying and optimizing the stochastic simulation of rare biochemical events

by

Max C Klein

A dissertation submitted to The Johns Hopkins University

in conformity with the requirements for the degree of

Doctor of Philosophy

Baltimore, Maryland

January, 2019

Abstract

The mechanics of complex biochemical systems, such as those responsible for genetic regulation in cells, can be understood by mapping their epigenetic landscapes. However, traditional deterministic simulation methods cannot be used to build these maps, while stochastic simulation methods are too slow, owing to the presence of rare events.

Enhanced sampling methods, such as forward flux sampling (FFS), hold a great deal of promise for accelerating stochastic simulations of nonequilibrium biochemical systems involving rare events. However, the description of the tradeoffs between simulation efficiency and error in FFS remains incomplete. We present a mathematically rigorous analysis of the errors in FFS that, for the first time, covers the contribution of every phase of the simulation. We derive a closed form expression for the optimally efficient count of samples to take in each FFS phase in terms of a fixed constraint on sampling error. We introduce a new method, forward flux pilot sampling (FFPilot), that is designed to take full advantage of our optimizing equation without prior information or assumptions about the phase weights and costs along the transition path. In simulations of both single- and multi-dimensional gene regulatory networks, FFPilot is able to perfectly control error due to undersampling. Once sampling error was controlled, we discovered that multidimensional systems have an additional source of error. We show that this extra error can be traced to correlations between phases due to roughness on the probability landscape. Finally, we show that in sets of simulations with matched error, FFPilot

is on the order of tens-to-hundreds of times faster than direct sampling, in a fashion that scales with the rarity of the events.

Acknowledgments

My deepest thanks to my parents, my advisor, my committee members, and all of my friends who bore with me during a difficult time. Your support has meant the world to me.

Table of contents

List of tables

List of figures

Chapter 1

Introduction

> *One of the great arguments of the day was vitalism versus mechanism, a disguised form of the old and continuing debate between those, including the religious, who believe the world has purpose and those who believe it operates automatically and by chance... The German chemist who scoffed in 1895 at the "purely mechanical world" of "scientific materialism" that would allow a butterfly to turn back into a caterpillar was disputing the same issue, an issue as old as Aristotle.*
>
> – Richard Rhodes, *Making of the Atomic Bomb*[1]

Are living organisms special? Are biological molecules inherently unique, or do they obey the same laws of chemistry and physics as every other molecule? Since the time of Mendel[2] and Cajal[3] the scientific consensus has been converging towards the latter view: life is made of the same material as the rest of the world. In other words, that the information encoded in an organism's genotype and (more broadly) biochemical state defines the organism's phenotype (*i.e.* it's appearance and behavior). Over the course of the 20th century this materialistic viewpoint has become an implicit assumption that underlies all current work done in the life sciences. However, this view is largely taken on faith, as a complete mechanistic link between biochemistry and behavior has been established for only a small subset of cellular systems. Though there is now an extensive (and ever growing) catalog of

the biochemical elements from which living systems are formed, the mechanisms by which they interact remain in general unexplained. The aim of this thesis is to present work towards a general method of mechanistically linking biochemistry to phenotype.

1.1 Models of biochemistry

The question at hand is one of dynamics. How does the biochemistry of a cell "unfold" over time to produce a phenotype*? Models, and modeling, form the cornerstone of a workable approach for explaining large-scale biological phenomena in terms of what goes on at the smallest scale. First, we need to establish a theoretical framework in which to reason about biochemistry. Any chemical reaction can be written in the form:

$$\alpha_1 a_1 + \alpha_2 a_2 + ... \longrightarrow \beta_1 b_1 + \beta_2 b_2 + ..., \tag{1.1}$$

where $\{a_1, a_2, ...\}$ and $\{b_1, b_2, ...\}$ are the reactants and products, and $\{\alpha_1, \alpha_2, ...\}$ and $\{\beta_1, \beta_2, ...\}$ are their respective stoichiometries. In some cases, a chemical species may appear on both sides of the reaction with the same stoichiometry, as so:

$$X \longrightarrow X + Y.$$

This implies that X is required for the reaction, but is not consumed by it.

To give a concrete example, consider protein expression from a gene. A simple model of this process can be derived from the central dogma[4,5]: say there is a segment of DNA that comprises the gene g_p. g_p is transcribed to produce an RNA rna_p. In turn, rna_p is then translated to a protein p. Both rna_p and p will degrade over time. The reactions that describe this system can be written in terms of Eq 1.1

*Equivalently, imagine that you are given a perfect description of the contents of a cell at one moment in time. What can you then say about the contents of that cell at some moment in the future? What can you say about the cell's behavior during all of the moments in between?

as so:

$$\begin{gathered} g_p \longrightarrow g_p + rna_p \\ rna_p \longrightarrow rna_p + p \\ rna_p \longrightarrow \emptyset \\ p \longrightarrow \emptyset \end{gathered} \quad . \tag{1.2}$$

In a sense, a list of reactions like Eq 1.2 gives a complete description of a system. This level of description can be thought of as forming a qualitative model. Qualitative models can be used to answer questions about what is possible in a system, or how a particular event occurs.

The same approach used to come up with Eq 1.2 can also be used to build more complex, regulated models of expression. For example, say that p is able to dimerize, and that p_2 dimers are able to bind back to g_p and repress further transcription. The reactions that describe the system would then be:

$$\begin{gathered} g_p \longrightarrow g_p + rna_p \\ rna_p \longrightarrow rna_p + p \\ p + p \longleftrightarrow p_2 \\ p_2 + g_p \longleftrightarrow p_2 \cdot g_p \\ rna_p \longrightarrow \emptyset \\ p \longrightarrow \emptyset \end{gathered} \quad . \tag{1.3}$$

1.2 Epigenetic landscapes and phenotypes

Low-level models of biochemistry such as Eq 1.2 can be linked to higher-level phenomena via the concept of the epigenetic landscape. Waddington introduced his

idea of the epigenetic landscape[6,7] by first asking the reader to imagine a cell as represented by a high-dimensional space. We call this space the cell's state space. Each dimension of the state space has a one-to-one correspondence with one of the species of metabolites or biomolecules present in the cell. The (non-negative, integer valued) coordinate along each dimension is the count of the corresponding molecule. Thus, each point in the complete state space uniquely defines a possible state of the cell. Just as there can be thousands, or tens of thousands, of distinct chemical species present in a cell, so too can a state space have thousands of dimensions.

The possible states of a cell are combinatorially vast. Simply enumerating them all would be a herculean computational task. Thus, it is legitimate to ask: what insight about a cell can actually be gained by thinking about it in terms of its state space? An analogy to the study of protein folding is useful here. Levinthal's paradox[8] says that there are more possible states for a protein to be in than atoms in the universe. Yet somehow proteins still fold. Similarly, cells tend to remain in homeostasis with respect to a particular phenotype (or sometimes to transition from one phenotype to another in an orderly fashion). In both cases, physical forces conspire to limit the occupied states to restricted regions. The epigenetic landscape is a description of the states that tend to be occupied, and of the forces that drive a cell to those states. A single "neighborhood" of states on the landscape constitutes a phenotype.

Waddington originally developed the idea of epigenetic landscapes as a way to explain the constrained diversity he observed in developing organisms. "Diversity" in the sense that embryonic cells have many possible end states (in terms of the cell type of their lineage in the mature organism), and "constrained" in the sense that even harsh chemical perturbations often shift those end states only slightly. As he described it, the landscape of a developing cell is like rough, hilly terrain

that is covered in divots connected by valleys. The divots represent the various possible phenotypes, and the valleys the transition pathways in between them. The developing cell, then, is like a ball that starts at a a high point on this terrain (see Fig 1.1). The robustness of the development process at any given moment can be thought of as analogous to the steepness of the landscape in the cell's immediate vicinity. As the cell (or its lineage) travels downhill, it proceeds through various phenotypes and the branching paths in-between. Eventually it reaches the bottom of the hill, along with its terminal cell type.

In the decades since Waddington first proposed it, much work has gone into filling in the details of the theory behind epigenetic landscapes. In the modern view[9,10,11], the epigenetic landscape is a representation of the physics of a non-equilibrium system. It can be (approximately) split into two parts: a potential surface (analogous to the potential energy surface of an equilibrium system) and a probability flux. If the potential surface is equivalent to Waddington's rough terrain, then the probability flux is like a strong wind blowing across it. Because of this "wind", a cell's fate is not determined by its potential surface alone. In recent years epigenetic landscapes of various cellular systems, such as the cell cycle[12,13] and oncogenesis[14,15], have been constructed. These high-dimensional landscapes can be plotted (via projection) in terms of one or more chemical species of interest. This gives a quantitative picture of how transitions in the population count of any given species drives transitions between phenotypes (see figure Fig 1.2).

1.3 Quantitative models and networks

Given that an epigenetic landscape can be used to link biochemistry to phenotype, the question then is how to calculate one. A qualitative model is not enough. What is needed is a quantitative model, one that can reproduce the actual species counts. In order to build a quantitative model of biochemical state and behavior, 4 pieces

of information are required:

1. The identity of the distinct interacting biochemical elements. These are often referred to as chemical species, or just species.

2. The list of reactions in which the chemical species participate.

3. The rate laws that describe how quickly each reaction occurs.

4. The quantity of each species present in the system. These are dynamical systems under consideration, so precise quantities can only be spoken of with respect to a particular moment in time (such as an initial time $t = 0$).

Once these 4 pieces have been determined, the next step is to combine them into a cohesive, mathematically tractable model. The standard way to do so is to build a network.

There are many different representations of biochemical networks, each with their own strengths and weaknesses. A simple network representation of our protein expression model can be seen in Fig 1.3. Here, the protein expression model is shown as a digraph (directed graph). Like any network, a digraph consists of a set of nodes and connecting edges. Additionally, each edge in a digraph encodes a direction, such that they start at a reactant node and end at a product node.

Simple digraph representations are useful since they offer an intuitable picture of the lists of species and reactions that make up a model. Another useful network representation is the Petri net. Originally developed by Petri in the 1960's[16] to model linked chemical reactions, Petri nets offer a natural way to model agent-based processes in general*. Petri nets were first applied to problems in systems biology around the turn of the century[17], and have since been extensively developed for biological applications[18,19,20]. Though less easy to understand at a glance

*When applying Petri nets to biochemical systems, the chemical species are the agents. This is in the sense that each molecule acts independently.

than simple digraphs, Petri nets have the virtue of being able to encode unambiguous descriptions of biochemical systems. Due to this formal rigor, once set up they can be mechanically transcribed into various mathematical forms that can then be used as input for many different simulation techniques.

Petri nets are bipartite digraphs."Bipartite" refers to the fact that every Petri net contains two distinct sets of nodes: one set, called the places, represent chemical species, and the other, called the transitions, represent chemical reactions. For every reaction that a species participates in as a reactant there is an edge that starts at the corresponding place and ends at the corresponding transition. Similarly, for every product a reaction creates there is an edge that starts at the corresponding transition and ends at the corresponding place. The weight of each edge is the stoichiometry of the attached species (whether as product or reactant) with respect to the attached reaction. Additionally, a Petri net has a marking, a list that contains the initial quantities of each species.

Figure 1.4 shows a Petri net representation of the simple expression model of Eq 1.2. In addition to the graphical representation, a Petri net can be uniquely specified as a set of matrices $\{P,\ T,\ M,\ Pre,\ Post\}$[21]. For the simple expression model these matrices would be:

$$P = \begin{pmatrix} g_p \\ rna_p \\ p \end{pmatrix},\ T = \begin{pmatrix} \text{transcription} \\ \text{translation} \\ rna_p \text{ decay} \\ p \text{ decay} \end{pmatrix},\ M = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix},$$
$$Pre = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},\ Post = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \tag{1.4}$$

where P is the list of places/species, T is the list of transitions/reactions, M is the list of markings corresponding to P, Pre is a $M \times N$ matrix (where M is the count of reactions, and N is the count of unique species) in which Pre_{ij} gives the weight of the edge connecting the ith species to the jth reaction (equivalently, the reactant

stoichiometry), and $Post$ is the same thing as Pre except that $Post_{ij}$ gives the weight of the edge connecting the jth reaction to the ith species.

For the more complex regulated expression system described in Eq 1.3, the equivalent $\{P,\ T,\ M,\ Pre,\ Post\}$ matrices would be:

$$P = \begin{pmatrix} g_p \\ rna_p \\ p \\ p_2 \\ p_2 g_p \end{pmatrix},\ T = \begin{pmatrix} \text{transcription} \\ \text{translation} \\ rna_p \text{ decay} \\ p \text{ decay} \\ p \text{ dimerization} \\ p \text{ dedimerization} \\ p_2 + g_p \text{ binding} \\ p_2 g_p \text{ unbinding} \end{pmatrix},\ M = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

$$Pre = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix},\ Post = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}. \tag{1.5}$$

Fig 1.5 shows a plot of the actual Petri net that the above matrices describe.

1.4 Deterministic simulation

1.4.1 ODE models of biochemistry

The traditional way to simulate biochemical systems begins with constructing a set of ordinary differential equations (ODEs). This type of simulation is commonly referred to as deterministic simulation (as opposed to stochastic simulation, which is discussed later in Sec 1.5). If there are N chemical species in a system, for each distinct species X_j there will be an equation of the form:

$$\frac{dX_j}{dt} = f_j\left(X_1, X_2, ..., X_N\right).$$

On the right hand side of the equation is a derivative that represents the rate in change over time in the quantity of X_j. On the left hand side is a function of the

quantity of every species (including X_j). The exact form of f_j will depend on the type (and precise mathematical formulation) of the chemical kinetics being modeled.

Given the Petri net representation of a system, it is straightforward to construct the set of ODEs. Say that the system involves N species participating in M reactions. First one constructs the M rate laws $r_i(X)$. The rate law $r_i(X)$ tells you how quickly each reaction i is occurring. Traditionally, the $r_i(X)$ are given a form based on mass action kinetics (which can be traced back to a set of papers published in the 1860s[22]). In the mass action view, the rate at which any reaction occurs is directly proportional to the quantity of each reactant. This implies that every reaction has an associated rate law $r_i(X)$ of the form:

$$r_i(X) = k_i X_1^{p_{i1}} X_2^{p_{i2}} ... X_N^{p_{iN}} = \prod_{j=1}^{N} X_N^{p_{ij}}, \tag{1.6}$$

where k_i is a rate constant, and p_{ij} is a shorthand for the ijth entry of the Pre matrix.

Next, one determines the stoichiometry matrix S. Essentially, the stoichiometry matrix is the difference of the $Post$ and Pre matrices. The exact definition of S varies in the literature, but for our purposes it will be convenient to define S as the transpose of the difference:

$$S = (Post - Pre)^\top.$$

Thus, S will be a $N \times M$ matrix. With the rate laws $r_i(X)$ and the stoichiometry matrix S in hand, the entire set of N ODEs can be written as a single matrix equation:

$$\frac{dP}{dt} = Sr, \tag{1.7}$$

where P is the places matrix, $\frac{dP}{dt}$ is a column vector of the ODEs, and r is a column vector in which the ith entry is the rate law $r_i(X)$.

The nature of the compact form given in Eq 1.7 is most easily explained with an example. For our simple expression system (described in Eq 1.2 and given in Petri

net matrix form in Eq 1.4), the column vector of rate laws r can be found from Eq 1.6:

$$r = \begin{pmatrix} k_1 \cdot g_p^1 \cdot rna_p^0 \cdot p^0 \\ k_2 \cdot g_p^0 \cdot rna_p^1 \cdot p^0 \\ k_3 \cdot g_p^0 \cdot rna_p^1 \cdot p^0 \\ k_4 \cdot g_p^0 \cdot rna_p^0 \cdot p^1 \end{pmatrix} = \begin{pmatrix} k_1 \cdot g_p \\ k_2 \cdot rna_p \\ k_3 \cdot rna_p \\ k_4 \cdot p \end{pmatrix}$$

and the stoichiometry matrix S is:

$$S = \left[\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} - \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right]^\top = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}^\top = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}.$$

The set of ODEs that describe the system is then:

$$\begin{pmatrix} \frac{d}{dt} g_p \\ \frac{d}{dt} rna_p \\ \frac{d}{dt} p \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} k_1 g_p \\ k_2 rna_p \\ k_3 rna_p \\ k_4 p \end{pmatrix} = \begin{pmatrix} 0 \\ k_1 g_p - k_3 rna_p \\ k_2 rna_p - k_4 p \end{pmatrix}.$$

Solving* the above system of ODEs yields:

$$\begin{pmatrix} g_p \\ rna_p \\ p \end{pmatrix} = \begin{pmatrix} M_1 \\ \frac{k_1 M_1}{k_3} + e^{-tk_3}\left(M_2 - \frac{k_1 M_1}{k_3}\right) \\ \frac{k_1 k_2 M_1}{k_3 k_4} + e^{-tk_3}\frac{(k_2(k_1 M_1 - k_3 M_2))}{k_3(k_3 - k_4)} + e^{-tk_4}\left(M_3 + \frac{k_2(k_4 M_2 - k_1 M_1)}{k_4(k_3 - k_4)}\right) \end{pmatrix}, \tag{1.8}$$

where M_j is the initial marking (*i.e.* quantity) of species j. For any given marking M and set of rate constants k_i, the expressions of Eq 1.8 can be used to find the quantity of each species at any given time t.

Often, a modeler is not concerned with the initial behavior of a system†, but only with the behavior of the system over long periods of time. In these cases, solutions such as those found in Eq 1.8 can be simplified by considering their value in the

*The complete details of solving a system of ODEs is outside of the scope of this thesis. In general, it suffices that a solution can easily be obtained using a computational algebra system such as Mathematica[23].

†As far as reproducing experimental results go, the initial behavior of a system is often irrelevant. When studying biological systems, it is typical for the system to preexist any experimental observation (though an obvious exception would be any stop-flow experiment).

limit $t \to \infty$, also called the steady state limit. The only terms in Eq 1.8 that explicitly depend on time are exponential decay terms of the form e^{-tx}. These exponential decay terms quickly approach 0 as t increases, yielding:

$$\begin{pmatrix} g_p \\ rna_p \\ p \end{pmatrix} \approx \begin{pmatrix} M_1 \\ \frac{k_1 M_1}{k_3} \\ \frac{k_1 k_2 M_1}{k_3 k_4} \end{pmatrix}.$$

1.4.2 The benefits and shortcomings of deterministic simulations

Deterministic, ODE based approaches such as the one given in the previous section were first used to model biological systems during the early 1900's[24,25,26,27]. Since then, ODEs have seen wide use in the simulation of biological systems. Most such systems of ODEs do not have a closed-form solution as in Eq 1.8. Instead, numerical integration[28] can be used to determine the quantity of each species up to a finite time. Even when numerical integration is required, ODE based methods tend to be some of the most computationally efficient and easy to implement techniques, contributing to their popularity.

Fig 1.6 shows a realization of an ODE simulation of our simple expression system. The expression in Eq 1.8 was used to find the concentration of protein at every time point up to ten hours. On the right hand side of the figure is a histogram of the count of protein along the time series. This can be thought of as an empirical estimation of the epigenetic landscape of the simple expression system. Thus, the ODE simulation predicts that the landcape of this system effectively consists of a single state at protein count $= 50$.

If we were to repeat this simulation, the exact same result would be produced. This highlights a key feature of ODE simulations: they're deterministic, in the sense that, given a particular system and set of initial conditions, the method will always produce exactly the same result (up to the accuracy of any numerical integration

method that was used). It is because of this reproducibility that ODE simulations are also called deterministic simulations.

The theoretical formulation of the ODE approach relies on a number of assumptions[29]. In particular, every chemical species involved in a reaction is assumed to be distributed across the reaction volume in a completely homogenous fashion. The concentration of any given species at any point in space is equal to the concentration of that species at any other point. This is in effect a continuum view, in which chemical species are not made of discrete particles but instead can be treated as infinitely divisible interacting fluids.

In the words of van Kampen, at the scale of test tubes the continuum assumptions are ”actually better obeyed than might be expected”[30]. At significantly smaller scales, such as those of a single cell, these assumptions break down. At some point it becomes no longer reasonable to assume that a chemical substance is infinitely divisible . Instead, one begins to have to account for effects[31] arising from the true, single molecule nature of chemical substances. In addition to small scales, these single molecule effects also become prominent when a substance is present in a very low concentration. In this type of situation it becomes more appropriate (and more accurate) to measure quantities in terms of particle count, also sometimes called copy number.

The assumptions of ODE modeling break down to some extent when applied to any gene expression system[32]. This is due to the fact that the DNA of each gene is typically present at very low copy numbers (just 2-4 for most genes in a diploid organism) in each cell. Moreover, many of the other components of expression systems, such as specific RNAs and proteins, also have low copy number. Thus ODE models cannot recapitulate the full behavior of gene expression. Even for our simple expression model, deterministic simulation fails to reproduce many of its details (as shown in Fig 1.7 and discussed in Sec 1.6). In order to capture the

full dynamics through which the information encoded in genes becomes proteins, alternative methods have to be used.

1.5 Stochastic simulation

> *The justification for using the stochastic approach, as opposed to the mathematically simpler deterministic approach, was that the former presumably took account of fluctuations and correlations, whereas the latter did not... in the deterministic formulation no distinction is made between the average of a product and the product of the averages; i.e., it is automatically assumed that* $\langle x_i x_j \rangle = \langle x_i \rangle \langle x_j \rangle$*. For* $i \neq j$ *this assumption nullifies the effects of correlations, and for* $i = j$ *it nullifies the effects of fluctuations.*
>
> – Daniel T Gillespie, *A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions*[33]

Stochastic simulation is a particularly robust method for recreating the behavior of interacting biomolecules. The stochastic simulation method stems from work done by Gillespie. In his seminal 1976 paper "A general method for numerically simulating the stochastic time evolution of coupled chemical reactions"[33], Gillespie laid out the theoretical framework of stochastic chemical kinetics (which was still in its infancy[34,35,36] at that point) and proposed the first practical algorithm for simulating stochastic chemical systems.

1.5.1 The theory of stochastic simulation

The essential assumption of stochastic chemical kinetics is that for every reaction R_i there exists a constant c_i such that[33]:

$$c_i dt \equiv \text{average probability that any set of competent* molecules react per } R_i \text{ during the time interval } (t, t+dt]. \quad (1.9)$$

It can be shown[37] that for any well stirred system that is also at thermal equilibrium, the condition in Eq 1.9 will be true for any reaction that follows mass-action kinetics.

The continuum chemical kinetics used in deterministic simulations assumes that each chemical species is homogeneously distributed across the reaction volume under study. In this view, each substance is in some sense spread infinitely finely over the volume. In stochastic kinetics, this homogeneity assumption is recast as the well-stirred assumption, in order to account for the existence of discrete molecules. Now it is assumed that only the probability distribution of each species is uniformly distributed over the volume. This means that in the stochastic view, at any given moment each particle of each species in a volume is equally likely to be anywhere within that volume. It is reasonable to assume that a system is well-stirred if it is small enough and if non-reactive collisions greatly outnumber collisions that result in a reaction. This description happens to describe most cellular compartments reasonably well. The fact that water molecules tend to vastly outnumber all other kinds within a cell is enough to satisfy the ”non-reactive collisions” assumption[33].

Building on Eq 1.9, a propensity function (referred to by some sources as a hazard equation[21]) can be defined[33] for each reaction R_i:

$$a_i(X)\, dt \equiv \text{probability that during the time interval } (t, t+dt] \text{ an } R_i \text{ reaction will take place,} \quad (1.10)$$

where X is a vector of the species counts $(X_1, X_2, \ldots, X_N)$. The propensity function

of any reaction following mass action kinetics can be defined as:

$$a_i(X) = c_i h_i(X), \tag{1.11}$$

where $h_i(X)$ is the count of sets of molecules that can participate in reaction R_i. Given the relevant Petri net matrices, a propensity function from the family defined by Eq 1.11 can be constructed for each of the M reactions of a system with N species by means of a general formula[21]:

$$a_i(X) = c_i \prod_{j=1}^{N} \binom{X_j}{p_{ij}}, \tag{1.12}$$

where p_{ij} is the ijth element of the Petri net matrix Pre.

In theory, all systems can be modeled using the propensity functions for these four elementary reaction types:

$$\begin{aligned}
&a_i(X) = c_i, && \emptyset \xrightarrow{c_i} \text{product} && \text{(zeroth order)},\\
&a_i(X) = c_i X_j, && X_j \xrightarrow{c_i} \text{product} && \text{(first order)},\\
&a_i(X) = c_i X_j X_k, && X_j + X_k \xrightarrow{c_i} \text{product} && \text{(second order)},\\
&a_i(X) = c_i \frac{X_j(X_j - 1)}{2}, && 2X_j \xrightarrow{c_i} \text{product} && \text{(second order self)},
\end{aligned} \tag{1.13}$$

where the actual formulae in the left column were derived from Eq 1.12. Other reactions, such as those of order three or above, are considered nonphysical, since an instantaneous collision of any three molecules is extremely rare. Given a unit volume*, the first three propensity functions above are identical to the corresponding deterministic rate laws, *i.e.* $a_i(X) = r_i(X)$ and $c_i = k_i$. There is no deterministic rate law equivalent to the second order self propensity function. For large enough X_j there is a reasonable approximation:

$$r_i(X) \approx \frac{c_i}{2} X_j^2 = k_i X_j^2, \qquad \text{(deterministic second order self)}, \tag{1.14}$$

*see Wilkinson[21], chapter 6, for full details on converting between deterministic rate laws and stochastic propensities when dealing with real volumes.

where we've set $k_i = \frac{c_i}{2}$ above.

Technically, any reaction not in Eq 1.13 should be considered a compound reaction, and modeled by decomposition into 2 or more elementary reactions. In practice, however, it is not uncommon for modelers to use propensity functions with a wide variety of forms, such as Hill equations:

$$a_i(X) = \frac{X_j^h}{\kappa^h + X_j^h}, \qquad X_j \longrightarrow \text{product} \quad \text{(Hill equation)},$$

where h and κ are arbitrary constants. The use of such alternative propensity functions are then justified on an empirical basis (*e.g.* they reproduce a particular feature of the modeled system), rather than on the basis of stochastic kinetics per se[38].

1.5.2 The stochastic simulation algorithm

The Gillespie algorithm, or (as Gillespie himself refers to it[39]) the stochastic simulation algorithm (SSA), was first proposed by Gillespie in 1976[33]. Although the implementation of the algorithm has been improved[40,41,42,43] over the years, the algorithm itself remains (40 plus years later) the gold standard of stochastic simulation methods.

SSA treats the underlying chemical system as a Markov process. This means that the next state that a system occupies is determined (in a probabilistic sense) solely by its present state*. The probability of each possible next state can be calculated by means of probability expressions of the form[33]:

$$p(t', i|X)\, dt \equiv \begin{array}{l}\text{probability that the next reaction in the system takes} \\ \text{place in the instantaneous interval } (t+t', t+t'+dt] \\ \text{and is of the type } R_i.\end{array} \tag{1.15}$$

The goal of SSA is to generate trajectories (*i.e.* time series) of the underlying stochastic chemical system by means of an iterative cycle. First the the probability

*Although the framework is also flexible enough to allow for the presence of explicitly time-dependent factors, such as externally applied chemical driving forces

distribution $p\left(t', i|X\right)$ (described in Eq 1.15 above) is calculated, then a random sample is drawn from $p\left(t', i|X\right)$. The state of the system is updated based on the outcome of the random sample, then the calculation of $p\left(t', i|X\right)$ is updated based on the new state of the system, and the processes is repeated many, many times.

The direct method (DM) variant of SSA is based on the fact that $p\left(t', i|X\right)$ can be split into two independent distributions:

$$p\left(t', i|X\right) = p_1\left(t'|X\right) \cdot p_2\left(i|t', X\right)$$

Stochastic kinetic theory and basic probability can be used to derive expressions[33] for p_1 and p_2:

$$\begin{aligned} p_1\left(t'|X\right) &= Ae^{-At} \\ p_2\left(i|t', X\right) &= \frac{a_i}{A}, \end{aligned} \tag{1.16}$$

where

$$\begin{aligned} a_i &= a_i\left(X\right) \\ A &= \sum_{i=1}^{M} a_i\left(X\right), \end{aligned}$$

The actual DM algorithm can be though of as a "kinetic" Monte Carlo method[29] that iteratively chooses the next time and reaction based on the generation of two uniform random numbers in a loop:

1. Calculate the current value of the propensity functions $a_i\left(X\right)$ and their sum $A\left(X\right)$, given the current state (t, X) where t is the current time and X is a vector of the current species counts.

2. Sample p_1 and p_2 by drawing two uniform random numbers u_1 and u_2 from the unit interval:

   (a) Calculate the time until the next reaction t' as:

   $$t' = \frac{1}{A} \ln\left(\frac{1}{u_1}\right)$$

(b) Calculate the index of the next reaction i as:

$$i = \text{the first value of } i \text{ such that } \sum_{j=1}^{i} a_j > u_2 A$$

3. Update the current time as $t+t'$, update the current species counts as $X+S_i$, where S_i is the ith column of the stoichiometry matrix S_i (see Secs 1.3 and 1.4.1).

4. Write out any desired information about the state, then either terminate the simulation (given that an appropriate condition has been met), or continue by returning to step 1.

After running (and recording) many such iterations, the data collected can be thought of as an exact realization of one replicate from the ensemble of the modeled system. In real computational experiments, typically many such replicates are run and the data from them are combined to produce a final analysis. For example, the epigenetic landscape of a system can be calculated by simply binning (*i.e.* into a histogram) the species count data of one or more replicates*.

1.6 Deterministic vs stochastic

In general, deterministic simulation requires less computational effort than stochastic. So why should a computational scientist bother using stochastic simulation? The theoretical justifications discussed in the previous few sections can be put into

*Technically, binning the species counts only gives one part of the epigenetic landscape, the steady state probability distribution $p^{\text{SS}}(X)$. The other component of the landscape, the steady state fluxes, can be calculated by binning the number of times each reaction occurs. Alternatively, the steady state flux $\Phi^{\text{SS}}_{\mathcal{A},\mathcal{B}}$ from state $\mathcal{A}$ to state $\mathcal{B}$ can be calculated directly from the propensities, weighted by the steady state probability distribution:

$$\Phi^{\text{SS}}_{\mathcal{A},\mathcal{B}} = -\sum_i a_i(\mathcal{A}) p^{\text{SS}}(\mathcal{A}) + \sum_j a_j(\mathcal{B}) p^{\text{SS}}(\mathcal{B})$$

where the first sum is over the reactions that can transition $\mathcal{A} \to \mathcal{B}$, and the second sum is over the reactions that can transition $\mathcal{B} \to \mathcal{A}$. In the language of Sec 1.2, the distribution and fluxes can be likened to a rugged terrain and the winds that blow over it, respectively.

concrete terms by comparing deterministic and stochastic simulations performed on the same model. Even for very simple gene regulatory networks (GRNs) the difference in the level of detail captured by each simulation type is clear.

A comparison of deterministic and stochastic simulations of our simple expression system (see Eq 1.2) can be seen in Fig 1.7. The top panel shows results from a version of the simple expression system with a relatively high average protein expression level (around $\sim 5 \cdot 10^3$). Under these conditions, the time series produced by the deterministic and stochastic simulations (shown in the top left panel) are in reasonably good agreement. As well, the epigenetic landscape predicted by stochastic simulation (shown in the top right panel) is in good agreement with the landscape predicted by deterministic simulation (which will just be a single peak around $\sim 5 \cdot 10^3$).

On the other hand, the deterministic and stochastic simulations diverge in the limit of low protein expression. The bottom two panels of Fig 1.7 show simulations of low expression variants of the simple expression system. Though the mean expression levels of the different simulation methods match (all are around ~ 50 counts), the moment-to-moment behavior of the simulated systems do not. The deterministic simulations predict that the protein count will be constant. On the other hand, the stochastic simulations predict (correctly) that the protein count will fluctuate significantly about the mean. Further, the deterministic simulations predict that there is no difference between the systems depicted in the middle and bottom panels, while the stochastic simulations again correctly show that the fluctuations in the bottom variant will be significantly larger.

The simulations in Fig 1.7 are clear examples of one of the key weaknesses of deterministic simulation: it does not capture fluctuations correctly. These kinds of species count fluctuations are often referred to in the literature as “noise”[44]. This

is something of a misnomer, as cellular noise has been found to play an important role in (and sometimes be a primary driver of[45,46]) a wide variety of decision making[47,48] and developmental processes[49]. On a more immediately visible note, the bottom two panels of Fig 1.7 show the dominant role that noise can play in moulding the epigenetic landscape. Most of the landscapes of the simple expression system variants are roughly symmetrical. Alternatively, the large fluctuations present in the noisiest variant (at the bottom of the figure) skew its landscape towards larger counts, giving it a long tail. This long tail is a nice illustration of the complex, non-symmetrical behaviors that can emerge in the limit of low copy number, demonstrating the need for a stochastic approach to the simulation of even the simplest GRNs.

If a system includes at least one nonlinear reaction (*i.e.* a reaction for which the corresponding rate law/propensity is nonlinear), it becomes possible for the results of deterministic and stochastic simulation to diverge completely. When working with nonlinear systems, deterministic simulation is in general unable to reproduce even the correct mean behavior[50]. Both the p dimerization and the $p_2 + g_p$ binding reactions of our self regulating expression system (see Eq 1.3) are second order, and thus nonlinear. Fig 1.8 shows the results from a deterministic and a stochastic simulation of this system. The deterministic and stochastic means do indeed differ signficantly, by $\sim 15\%$.

Regardless of the ability of deterministic simulation to reproduce the correct mean behavior, a larger issue remains. The self regulating expression system depicted in Fig 1.8 has more than one metastable (*i.e.* long-lived) state. Evidence for this can be seen in both the time series (*e.g.* the long stretch around hour 6 during which the protein count stays fixed near 0) and via the fact that its epigenetic landscape has more than one mode/peak. Deterministic simulation cannot be used effectively to map the various states of this kind of multi-stable system,

nor to calculate the dynamics of the transitions in between them. Thus, in order to construct the epigenetic landscape of a complex, nonlinear system with multiple possible states, stochastic simulation is required.

1.7 Rare events, statistics, and the limits of stochastic simulation

With respect to a particular system, an event is rare[51] if it occurs at a much slower rate than the fastest event*. In a multi-stable system, switching between states tends to be a rare event since it often requires the co-occurrence of specific fluctuations in two or more noisy reaction channels. For example, in order for the self regulating expression system shown in Fig 1.8 to switch from active to repressed, first a p dimerization reaction (itself a rare event) must occur. Then the $p_2 + g_p$ binding reaction must happen before the p_2 dimer dissociates.

The existence of a rare event implies a large separation between the signficant timescales of a system. This timescale separation can prove challenging[52,53] for many different simulation techniques. Theoretically, SSA simulation can exactly reproduce the behavior of any system, regardless of the presence of rare events. In practice however, it can be difficult, or even impossible, to produce an accurate simulation of a rare event systems using SSA. This apparent paradox can be understood by considering the error statistics of SSA.

When studying a rare event, it is standard practice to first determine its $MFPT$, the mean waiting time before the event occurs. Given a rare event, stochastic simulation can be used to determine its $MFPT$. The procedure is equivalent to estimating the mean of a random variable $\mathbb{w}$ (in this case, $\mathbb{w}$ is the waiting time in between occurrences of the event). A single sample $\mathbb{w}_i$ is drawn from $\mathbb{w}$ by

*How much slower? There is no formal definition of a rare event, but as a rule of thumb assume that in order to qualify as rare, an event must occur at least 3 orders of magnitude less often than the system's most frequent event

running a trajectory until the first occurrence of the rare event, then recording the final simulation time. n samples are drawn by running n such replicates. The mean of these samples:

$$\frac{1}{n}\sum_{i=1}^{n} \mathbb{w}_i,$$

is then an estimator of $MFPT$.

The accuracy of the $MFPT$ calculated by SSA depends on the size of the sample count n. The exact form of the dependence depends in turn on the exact form of $\mathbb{w}$ (though it can in general be said that the accuracy increases along with the sample count). For a rare switching event (*e.g.* a transition that takes a system between two well separated states $\mathcal{A}$ and $\mathcal{B}$), $\mathbb{w}$ will tend towards an exponential distribution[54], assuming that there are no intervening states. In the limit of large sample sizes, the margin of error of the stochastic $MFPT$ calculation is then given by a simple formula (see Sec 2.1.4 for derivations and a complete discussion):

$$\frac{z_\alpha}{\sqrt{n}},$$

where z_α is the z score for confidence level α (*i.e.* $z_{.95} \approx 1.96$). Fig 1.9 shows the margins of error for a wide range of sample count values. In particular, it shows that a very large (38415) sample count is required in order to ensure that a simulation produces an accurate (*i.e.* no more than 1% error) $MFPT$ value.

For simple enough systems with few enough components, the computational cost (in terms of CPU time) required for accurate stochastic simulation is merely an inconvenience. For complex systems involving rare events, the computational cost can be prohibitive. Very roughly, the cost of sampling a rare event using standard SSA will scale as:

$$\frac{\Phi_{\text{fastest}}}{\Phi_{\text{rare}}}, \tag{1.17}$$

the quotient of the fluxes of the fastest event and the rare event. For a rare enough event, the computational requirements of accurate simulation will easily eclipse the

resources provided by any single computer. When a single computer is insufficient, one approach is to scale up your simulations to make use of HPC/cluster resources (an example of this is shown in Fig 1.10). A better approach is to make use of enhanced sampling. Enhanced sampling is a family of simulation methods that scale more efficiently than Eq 1.17 with respect to rare events.

1.8 Enhanced sampling

> *We are here concerned with essentially the same problem that some of the other speakers have spoken about. We wish to estimate the probability that a particle is transmitted through a shield, when this probability is of the order of* 10^6 *to* 10^8*, and we wish to do this by sampling about a thousand life histories. It's clear that a straightforward approach will not give the results desired.*
>
> – Kahn and Harris, *Estimation of particle transmission by random sampling*[55]

So begins the first paper ever written on the topic of enhanced sampling*. The family of enhanced sampling methods can be traced back to work done by von Neumann[56] and a few others[57,58] in the 1950s, during the early days of Monte Carlo nuclear physics simulations.

Stochastic simulation, though potentially more rigorous and accurate than the alternatives, is grossly inefficient. The trajectories of stochastic simulations are bound to follow the probability distributions of their underlying systems, and so by definition waste all but a slim minority of their time fluctuating around the high-likelihood regions of state space. Many researchers (both in systems biology and

*Technically, Kahn et al., 1951[55] is one of two papers about enhanced sampling that were published in the same conference proceedings. However, Kahn starts on an earlier page than von Neumann, 1951[56], so I'll call Kahn the first.

in many related fields[59,60,61,62]) over the years have asked themselves, "why can't I just confine my simulations to the part of state space I actually want to see?"

The typical goal of running a stochastic simulation is to collect samples, and then use those samples to estimate the value of some system property that cannot be calculated directly. Certainly it is technically feasible to simply bias or confine a simulation so that it remains within a region of interest, producing more relevant samples more quickly. However, a naive attempt at placing one's thumb on the scales of a simulation will inevitably create a distorting bias; that bias will pass on to the samples taken during the simulation and ruin the statistical validity of any result. It is possible to both have your cake (confine your simulation to a region) and eat it too (produce an unbiased sample) by applying bias in a controlled fashion, while at the same time keeping track of any distortion you cause via bookkeeping. With the right information the collected samples can then be unbiased, producing the desired result. This is the essence of the enhanced sampling methods.

In the broadest terms, all enhanced sampling methods follow the same basic script:

1. Generate a biased version of the original system.
2. Draw samples from the biased system.
3. Based on bookkeeping information recorded during the preceding steps, recover an unbiased sample by applying statistical methods to the biased sample.

There are now dozens (if not hundreds) of different enhanced sampling methods, and while some of them may not strictly follow these steps, each and every one of them performs the 3 actions listed above in some fashion.

1.9 Enhanced sampling methods for stochastic simulation of biochemistry

Beginning with the proposal of FFS in 2005[63] by Rein ten Wolde and coworkers, a number of different enhanced sampling methods have been developed for use in stochastic simulations of biochemical systems. Along with FFS, nonequilibrium umbrella sampling (NEUS) and weighted ensemble (WE) are also highly cited methods.

FFS, NEUS, and WE have a number of features in common. Each method requires the user to specify a function of their system's complete state to use as an order parameter:

$$\mathcal{O}\left(t, X\right) = \left(o_0, o_1, \cdots\right)$$

Additionally, each requires the user to specify a region of interest in terms of a set of bins that span an interval along the order parameter. Taken together, the order parameter and the tiling define a constraint that will be used to bias the user's system. The order parameter defines the degrees of freedom of the constraint, while the tiling defines one or more bounded regions along those degrees of freedom. Ultimately this constraint is used to guide the system into and along the region of interest, though the actual implementation of the constraint varies from method to method.

1.9.1 Forward flux sampling

The first step of FFS is to define two points of interest in your system's state space (we'll call these $\mathcal{A}$ and $\mathcal{B}$), a 1D order parameter that can unambiguously differentiate those two points, and a 1D sequence $\left(\lambda_0, \lambda_1, \ldots, \lambda_N\right)$ of "interfaces" (*i.e.* bins) that lie along the order parameter in between $\mathcal{A}$ and $\mathcal{B}$. If a trajectory passes below $\lambda_{\mathcal{A}} = \lambda_0$ it is said to be in the $\mathcal{A}$ state, and if it passes above $\lambda_{\mathcal{B}} = \lambda_N$ it is said

to be in the $\mathcal{B}$ state. FFS also requires the user to manually specify a maximum simulation time T and a set of trajectory counts $(n_1, n_2, \ldots, n_N)$, the significance of which are discussed in the following paragraphs.

Given the above setup, the goal of an FFS simulation is to calculate $\Phi_{\mathcal{A},\mathcal{B}}$, the flux from state $\mathcal{A}$ into $\mathcal{B}$. $\Phi_{\mathcal{A},\mathcal{B}}$ can be decomposed into a smaller flux and a probability term:

$$\Phi_{\mathcal{A},\mathcal{B}} = \Phi_{\mathcal{A},0} P\left(\lambda_{\mathcal{B}} \mid \lambda_0\right),$$

where $\Phi_{\mathcal{A},0}$ is the flux from the vicinity of $\mathcal{A}$ to the initial interface λ_0, and $P\left(\lambda_{\mathcal{B}} \mid \lambda_0\right)$ is the probability that a trajectory will reach $\mathcal{B}$ before it falls below λ_0, given that the trajectory is currently at λ_0. The probability can be further decomposed into a sequence of probabilities along the interfaces:

$$\Phi_{\mathcal{A},\mathcal{B}} = \Phi_{\mathcal{A},0} \prod_{i=1}^{N} P\left(\lambda_i \mid \lambda_{i-1}\right). \tag{1.18}$$

The actual FFS simulation consists of an ordered sequence of $N+1$ distinct phases. A schematic illustration of these phases is shown in Fig 1.11. Each phase produces an estimate of one of the terms on the left hand side of Eq 1.18.

The simulation begins with phase 0, at the start of which a single unbiased trajectory is initialized at $\mathcal{A}$ and allowed to run until the user-specified simulation time T. Whenever this trajectory crosses λ_0 while traveling forward (*i.e.* towards $\mathcal{B}$) the species counts at the point of crossing are recorded in a list C_0. If the trajectory ever leaves state $\mathcal{A}$ (*e.g.* by crossing into $\mathcal{B}$), the simulation time is paused, and doesn't resume incrementing until the trajectory reenters $\mathcal{A}$. The phase 0 trajectory is terminated once it reaches time T, and the flux term in Eq 1.18 is then estimated as:

$$\Phi_{\mathcal{A},0} = \frac{n_0^s}{T},$$

where n_0^s is the count of entries in C_0. Phase 0 then ends, and phase 1 begins.

During each phase $i > 0$, a user-defined number n_i of trajectories are run. Each

trajectory is initialized at a randomly chosen point from C_{i-1}, the list of crossing points from the previous phase. The trajectory is run until it either falls below λ_0 or passes above λ_i. If the trajectory passed above λ_i its species count at the point of crossing is recorded in C_i, and in either case the trajectory is terminated. Once n_i trajectories have run, the corresponding probability term in Eq 1.18 is then estimated as:

$$P\left(\lambda_i \mid \lambda_{i-1}\right) = \frac{n_i^s}{n_i}.$$

Phase i then ends, and $i+1$ begins. Once phase N finishes, the FFS simulation is over. The flux $\Phi_{\mathcal{A},\mathcal{B}}$ can be calculated from Eq 1.18, and the $MFPT$ can be calculated as the inverse flux, ${\Phi_{\mathcal{A},\mathcal{B}}}^{-1} = MFPT$.

Various extensions for the FFS method have been developed. One of the more useful ones is from a follow-up paper[64] that showed how FFS can be used to calculate a wide variety of values beyond flux and $MFPT$. The paper defines a set of weights that can be used to combine biased data collected during the individual phases into a single set of unbiased results, equivalent to those that could be gained from an unconstrained SSA simulation. Among other things, this enables the use of FFS to rapidly construct the epigenetic landscape of a system.

A more complete description of FFS (including a including a step-by-step listing of the algorithm) is given later in this thesis in Sec 2.1.1. The version given in Sec 2.1.1 differs somewhat from the original implementation of FFS, with each change made either for the sake of parallelizing the algorithm, or to make one of the method's outputs more amenable to statistical analysis. On the other hand, the description given above is faithful to the version from the originating papers[63,65,66].

1.9.2 Nonequilibrium umbrella sampling

NEUS was proposed by Dinner and coworkers[67] in 2007, a few years after FFS was introduced, and bears it a number of similarities. Like FFS, NEUS requires

an order parameter, as well as a set of interfaces that divide it into bins. It too stops trajectories when they intersect with those interfaces, and it too starts new trajectories based on a list of the points at which those trajectories intersected.

On the surface, there are also many differences between the two methods. In contrast to FFS, which requires the use of a 1-D order parameter, NEUS allows for the use of 2- and higher-D order parameters/interfaces. The reason why NEUS is able to make use of these more complex order parameters is in turn due to differences in how it implements the starting and stopping of trajectories.

Fig 1.12 shows a schematic of a simple case of NEUS simulation, in which a state space has been divided into 4 regions along a 2D order parameter. A NEUS simulation proceeds with respect to a given set of regions as follows*:

1. An initial weight term $w_i = \frac{1}{N}$ is assigned to each region i (where N is the count of regions), and a trajectory is started in each region.

2. Each trajectory is allowed to run unconstrained until it reaches one of the edges of its region. Given that the trajectory in region Λ_i reached the boundary between Λ_i and some other region Λ_j, two things happen:

   (a) The trajectory is terminated.

   (b) w_i is decreased and w_j is increased, according to the equation:

   $$-\Delta w_i = \Delta w_j = s w_i,$$

   where Δ implies an additive change, and s is a small constant factor.

3. Whenever a trajectory in a region Λ_i is terminated, a new trajectory is started

*Curiously, Dinner and coworkers have never published a full step-by-step listing of their NEUS algorithm. On the other hand, Vanden-Eijnden and coworkers published a very complete implementation of NEUS[68], though with a few of their own modifications (and, also curiously, they never actually use the name NEUS).

at one of the states α contained within Λ_i. α is selected at random with probability:

$$p_\alpha = \frac{\Phi_\alpha}{\sum_\alpha \Phi_\alpha},$$

where Φ_α is the total flux into state α from all states not contained in Λ_i. The flux terms Φ_α are calculated as:

$$\Phi_\alpha = \sum_\beta \frac{n_{\alpha\beta} w_\beta}{T_\beta},$$

where the sum is over all of the states β that are adjacent to α but outside Λ_i, $n_{\alpha\beta}$ is the observed count of crossings from β into α, w_β is the weight of the region that contains β, and T_β is the total simulation time of all trajectories run in the region containing β.

Over time, the weight terms w_i will converge[69]. At the end of a NEUS simulation, the steady state probability distribution of a system can be easily calculated as:

$$P(\alpha) = \sum_i P_{\Lambda_i}(\alpha) w_i,$$

where the $P_{\Lambda_i}(\alpha)$ terms are the probability distributions seen by the trajectories running in each separate region. These can in turn be calculated as:

$$P_{\Lambda_i}(\alpha) = \frac{t_\alpha}{T_{\Lambda_i}},$$

where T_{Λ_i} is the total simulation time (across all trajectories) spent in region Λ_i, and t_α is the total simulation time spent in state α.

On the other hand, NEUS originally could not be used to calculate $MFPT$ values. The developers of NEUS claimed that FFS and NEUS were orthogonal in their strengths and applications: FFS could be used to get rates like $MFPT$, whereas NEUS could be used to get steady state distributions. However, while the initial NEUS paper was in press, the developers of FFS published a new paper showing that their method could in fact also be used to calculate steady state distributions[64].

Symmetrically, a few years later a paper was published that showed how to use NEUS to find the $MFPT$[70].

In a later treatment[69] Dinner built on this convergence, and demonstrated that the differences between FFS and NEUS may be more superficial than they appear. He showed that, given appropriately defined regions, NEUS will actually completely reproduce the behavior, results, and statistics of FFS. From the viewpoint of Dinner's mathematical framework, FFS can in fact be thought of as a subset of the nonequilibrium umbrella approach.

1.9.3 Weighted ensemble

> *...one defines regions of importance in the space being studied, and, when the sampled particle goes form a less important to a more important region, it is split into two independent particles, each one-half the weight of the original. If you go from a more important to a less important region you double the weight... but play a game of chance, with probability one-half... to decide if the history is to be continued. The purpose of this is to spend most of the time studying the important rather than the typical particles, but to do it in unbiased fashion.*
>
> – Kahn and Harris, *Estimation of particle transmission by random sampling*[55]

Though the above quote is about Von Neumann's splitting technique, it could be to said to equally well describe the contemporary weighted ensemble WE method. As has been pointed out by several prominent WE researchers[71,72], WE is a strong echo of the earliest enhanced sampling work; it can be thought of as a rediscovery of the approach taken by Kahn, Von Neumann, and others during the 1950's.

Thus, half a century after its initial proposal, the splitting technique was reborn in the molecular dynamics community[59] as WE. More than a decade after it was

first proposed, a paper[73] was published that showed how WE could be adapted for use with stochastic biochemical simulations. Like FFS and NEUS, while at first the values that WE could be used to calculate were limited*, over time the method was further developed and refined.

Eventually, a version of WE was published that could be used to calculate both the $MFPT$ and the steady state distribution[74]. First, two states of interest $\mathcal{A}$ and $\mathcal{B}$ are chosen; between these states the system is split into a user-defined set of bins along an order parameter (which, like in NEUS can be multi-dimensional). Next, the actual simulations begin:

1. n trajectories are initialized in $\mathcal{A}$. Each is assigned a weight of $1/n$ and is allowed to run unconstrained.

2. Once a fixed time interval τ has passed, all trajectories are temporarily paused and each bin is visited (see Fig 1.13):

   (a) Skip the bins that currently have no trajectories in them.

   (b) While a bin has $0 < i < n$ current trajectories, pick a trajectory and duplicate it. When duplicated, each resulting "daughter" trajectories will inherit one half of the weight of the original.

   (c) While a bin has $i > n$ current trajectories, pick 2 trajectories, then terminate one of them. The surviving trajectory inherits the sum of the weights.

3. The trajectories are then resumed, and the simulations continue. Steps 2 and 3 are repeated until some termination condition is met.

At every simulation step, the total sum of trajectories weights will be 1. Over time, the WE algorithm will populate every bin with n trajectories, and the bin weights

*In the case of early WE, it could initially only be used to find the steady state probability distribution of a system.

(*i.e.* the sum of the trajectory weights in each bin) will converge to a steady state. Ideally, a WE simulation would terminate upon convergence of the bin weights, though in practice it is typical to end a simulation after a fixed time.

In comparison to other enhanced sampling (ES) methods, the WE algorithm is relatively simple. To an extent, the data it produces is also relatively simple to analyze. At the end of a WE simulation, the bin weights are literally an estimate of the the steady state probability distribution*. On the other hand, the calculation of $MFPT$ is significantly more involved. In brief, first the complete set of bin-to-bin transition rates is calculated by tracking every event in which any trajectory crosses from one bin to another; the rates are then combined into the overall $MFPT$ via a sum weighted by the bin weights[74,76,77].

This fits neatly into a trend that can be seen amongst all ES methods. In some deeper sense, all of the ES methods are the same[78,79]. With the right modifications, it seems that any ES method can be adapted to calculate any value that could be gained from unbiased simulation. As practical methods, however, it is clear that each have their individual strengths; FFS can be most easily and efficiently implemented to calculate $MFPT$ values, whereas for NEUS and WE are best suited for calculation of the steady state distribution. The true nature of these differences remains an open research question. By gaining a deeper and more rigorous understanding of the mathematics of ES, it is likely that the disparate existing methods can be unified on a theoretical level, and new, more efficient methods can be developed.

*The catch is that distribution estimate is given in "bin" coordinates. Nonetheless, since WE can be used effectively with unit spaced bins[75], this simple analysis can actually give a very accurate picture of the real steady state distribution.

1.10 Simplifying and optimizing forward flux

The original formulation of FFS[63] left a great deal of room for improvement. Specifically, FFS adds a large number of unspecified degrees of freedom to a system, in the form of a large set of extra simulation parameters. When a computational scientist sets up a FFS simulation, she must specify two points of interest, a 1D order parameter, a set of interfaces λ_i, the phase 0 maximum simulation time T, and the set of phase $i > 0$ trajectory counts n_i. These simulation parameters are required in addition to all of the model data required for a standard SSA simulation. In theory, the choice of FFS parameters will not have an effect on simulation outcome[66], only on simulation efficiency.

The additional simulation parameters present several challenges to users. For new users, the FFS parameters can be an imposing hurdle. They are likely one of the stumbling blocks preventing wider adoption of FFS methods. In particular, the order parameter and the interfaces are conceptually complicated. It takes time for a beginner to learn how to find reasonable values for these parameters. Even for an experienced user it can take a significant amount of trial and error[80] to find good values when working with a novel system. Further, for most systems there exists an optimal[81] set of FFS parameters that will maximize their efficiency and minimize their runtime.

It is possible to devise automated methods that can find and set the optimal FFS parameters values without user input. Several examples are discussed in the paragraphs below. These optimization routines kill two birds with one stone: they simplify the setup of an FFS simulation and make the method easier to use, while also speeding the simulation up. Between the published optimization methods and the work presented in this thesis (see Sec 1.11), it is possible to find optimized values of every FFS parameter (even the order parameter[82]). Thus, it should be

possible to devise a method that can automatically set an optimal value of every parameter all at once, though none has yet been published.

Along this vein, Borrero and coworkers devised a number of different optimization routines[83] for FFS. These routines are designed around the concept that the user's computational resources are limited. Thus, they require the user to input a quantity of computational effort, given in terms of the total number of trajectories that will be run during a single FFS simulation. The routines then find an optimal set of parameters that minimizes overall simulation error while holding the computational effort fixed at the user-specified level. For standard FFS, they devised one routine that can be used to find the optimal choice of n_i (the number of trajectories to run in each phase $i > 0$), and another that can be used to find the optimal choice of λ_i (the placement of the interfaces along the 1D order parameter). Since computational effort is held fixed, Borrero's optimization methods will not speed a simulation up, but will instead decrease the level of error in the simulation's results. Indeed, they found that their optimization methods could reduce simulation error by as much as 40%.

Borrero's interface optimization method generates an initial rough guess of λ_i and then using iterative rounds of FFS simulation in order to refine it. Kratzer and coworkers developed a more elegant alternative approach[80] to optimizing λ_i that avoids the need to perform extra simulations. Instead, they dynamically generate an optimal placement of interfaces, one at a time, during an otherwise standard FFS simulation. Kratzer describes two different variants of his method, but both follow roughly the same script: at the start of a simulation only the first and last interfaces, λ_0 and λ_N, are defined. At the start of each phase $i > 0$, a fixed number of trial trajectories are launched from λ_{i-1} under controlled conditions. Based on the outcomes of these trial trajectories, a location for λ_i is chosen, and the FFS simulation proceeds as normal (until the start of the next phase). Kratzer found

that his dynamic interface optimization methods were able to speed simulations up by as much as 2X, relative to a simulation run with manually placed interfaces.

The optimization schemes of both Borrero and Kratzer ignore the contribution of phase 0 to the simulation error. Technically, they both treat the outcome of phase 0 as deterministic, causing it to "fall out" of their analyses of the simulation variances and errors. This is standard practice[84] in analyses of the error statistics of FFS. We devised a novel approach to the analysis of FFS error that allows the contribution of phase 0 to be calculated directly (see Sec 2.2.1.1). Contrary to previous claims[66] in the literature, it can be shown that phase 0 can be a large contributor to overall simulation error (see Secs 2.2.4 and 2.2.5).

1.11 Forward flux pilot sampling

In many ways enhanced sampling seems too good to be true. It seems kind of like something for nothing, or a free lunch. Though the developers of various enhanced sampling methods have claimed that they are faster than direct sampling while introducing no extra approximations or errors[75,84], it remains to be definitely proven one way or the other. Thus it is reasonable to ask "does enhanced sampling actually work? Will it actually produce an answer of equivalent accuracy in less time than the established methods?" More prosaically, when performing FFS simulations, a researcher might wonder "how many trajectories should I run in each phase in order to get a result without too much error?" All of these are questions that the work presented in the subsequent chapters of this thesis attempts to answer.

In chapter 2 we present a new analysis of the error in the output of FFS simulations. Using a novel derivation, we find a more general form of the FFS error relation (*i.e.* simulation error as a function of the simulation parameters) than has been presented in the literature[84,83,66]. In particular, this allows us to calculate the previously unconsidered error arising from phase 0. We show that this phase 0

error can indeed be a significant contribution to the overall simulation error.

Next, we derive an equation that gives the optimal (in terms of computational cost) number of trajectories to run in each phase, given a user-defined desired maximum level of error in the simulation results, which we call an error goal. Based on this optimizing equation, we develop a novel variant of the FFS enhanced sampling method which we call FFPilot. FFPilot replaces the T and the entire set of n_i parameters of standard FFS (*i.e.* the parameters that determine the computational effort expended during each phase) with a single error goal parameter. Given that error goal, FFPilot will plan out and run the fastest possible FFS simulation. Thus, FFPilot is both a simplification and an optimization of the FFS approach. We wrote an optimized, fully parallelized implementation of the FFPilot alogorithm in C++. This implementation of FFPilot was added to Lattice Microbes[85], a stochastic simulation software package published by the Roberts Lab.

The remainder of chapter 2 is dedicated to a thorough validation and exploration of FFPilot. In simulations of 1D biochemical systems (*i.e.* systems with only a single chemical species), we show that FFPilot is indeed able to control error in the final simulation results as expected. In fact, FFPilot controls error so well that it is able to uncover a previously invisible problem in all existing analyses of FFS error, including my own. The published error analyses[84,83,66] all assume that sampling error (*i.e.* error due to running too few trajectories) is the sole non-negligible source of simulation error. However, when using FFS to simulate a complex system with a rough, multidimensional epigenetic landscape, it turns out that other sources of error can become significant. Results from FFPilot simulations of multidimensional systems show that while sampling error is still the dominant source of error, there is an anomalous extra error that FFPilot is unable to control. During each phase, a list of trajectory end states is built up and then used to initialize new trajectories during the next phase. We show that correlations exist between these lists, and we

demonstrate that these correlations are a necessary and sufficient cause for the anomalous error.

Chapter 3 is a brief conclusion. It includes a discussion of future improvements and possible directions for FFPilot.

The final chapter of the thesis is a tutorial that explains in detail how to use the FFPilot implementation in Lattice Microbes. Complete examples are given that show how to use FFPilot to calculate both the $MFPT$ and the epigenetic landscape of a system.

Figures

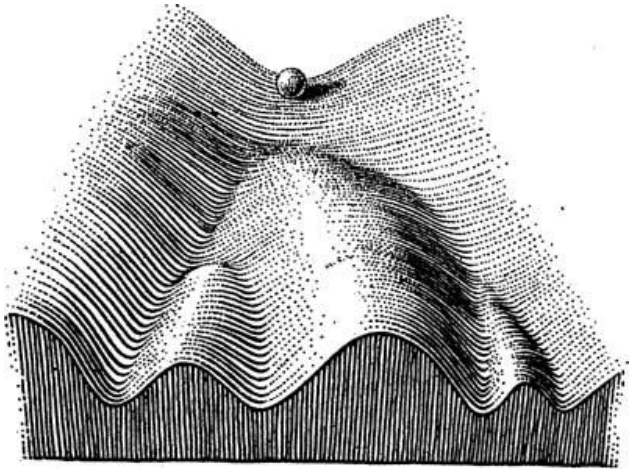


Figure 1.1: The epigenetic landscape of a developing cell, as envisioned by Waddington. The ball represents a cell at the beginning of development. Eventually the ball/cell will reach the bottom of the hill, ending up in one of several possible terminal cell types. The steep terrain counteracts most perturbations by forcing the ball back onto the path, making the whole process robust. Reprinted from Waddington, 1957[7].

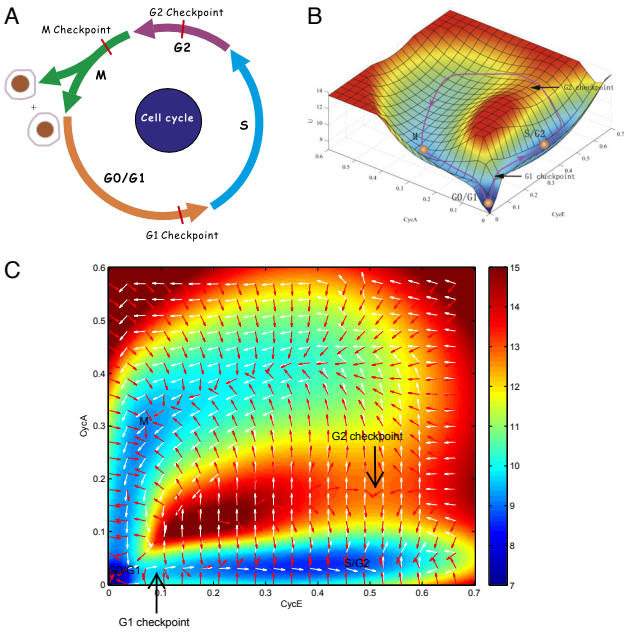


Figure 1.2: Three views of the mammalian cell cycle. A) The classical schematic view. B) The potential surface of the cell cycle. Data was taken from a 44 dimensional model and projected onto the concentrations of two cyclins, $CycA$ and $CycE$ (arbitrary units). The low points on the surface correspond to the various phenotypes along the cell cycle, and the transition path between them is shown. C) A 2D rendition of the complete epigenetic landscape of the model shown in B. The red arrows show the gradient of the potential, whereas the white arrows show the probability flux. Note that they never exactly coincide. Reprinted from Li et al., 2014[12].

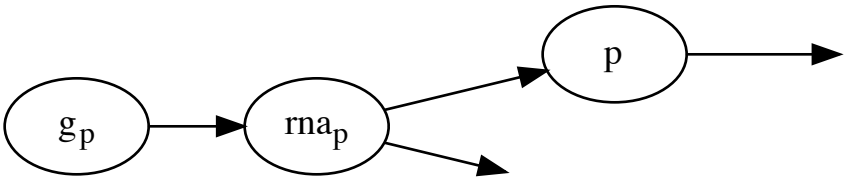


Figure 1.3: A simple digraph representation of the simple gene expression model from Eq 1.2. An arrow from one species to another implies that the first species is involved in the production of the second. Arrows to empty space imply a decay reaction.

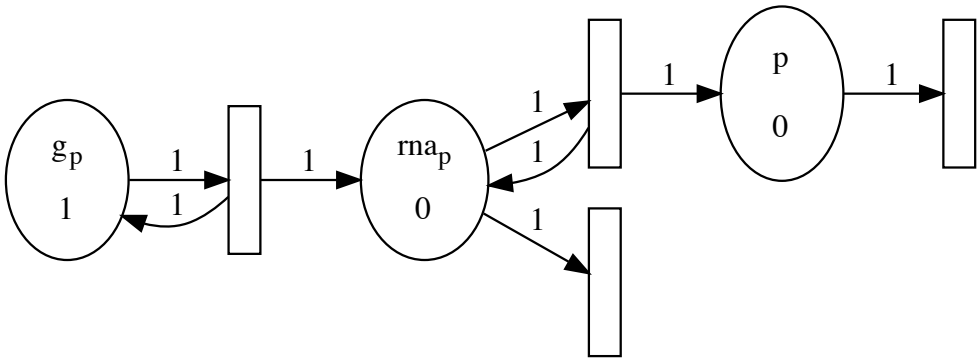


Figure 1.4: A Petri net representation of the simple genetic expression model from Eq 1.2. Arrows starting at places (round nodes) and ending at transitions (rectangular nodes) imply that the connected species is a reactant in the connected reaction. Arrows starting at transitions and ending at places imply that the connected reaction produces the connected species. Above each arrow is a weight that gives its stoichiometry. The marking (the count of each species) is shown underneath the name of each species. Transitions that have no arrows leaving them imply a decay reaction.

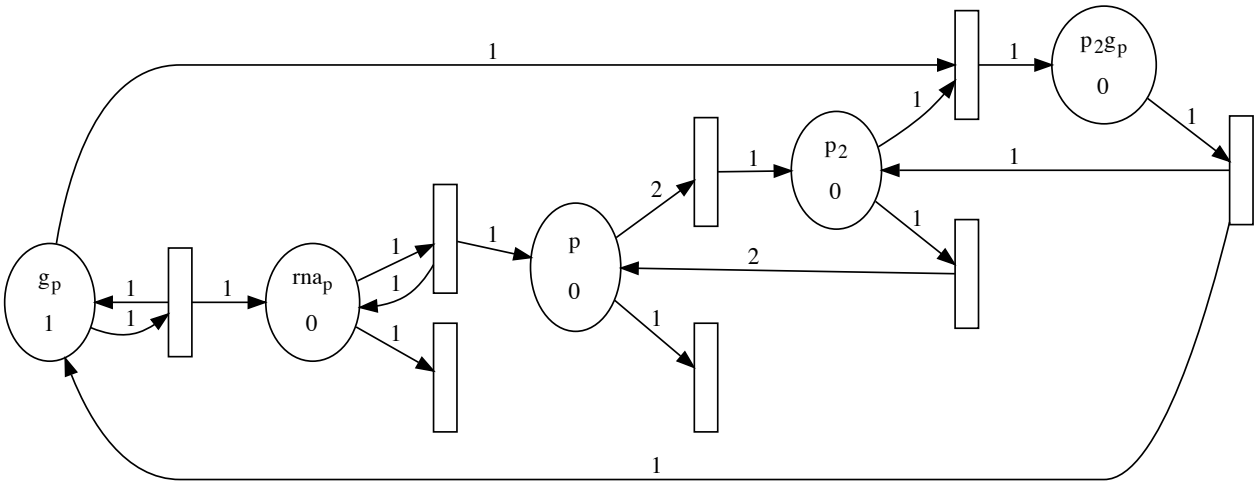


Figure 1.5: A Petri net representation of the self regulating expression model from Eq 1.3. The network is represented as described in Fig 1.4. The self regulating model extends the simple model (formally, the set of nodes and edges that describe the self regulating model are a proper superset of the nodes and edges of the simple model), and adds a set of reactions that combine to exert a negative feedback on the production of protein p. The reactions that directly drive this negative feedback are illustrated above in the 4 edges that encircle the graph and connect the g_p and $p_2 g_2$ species.

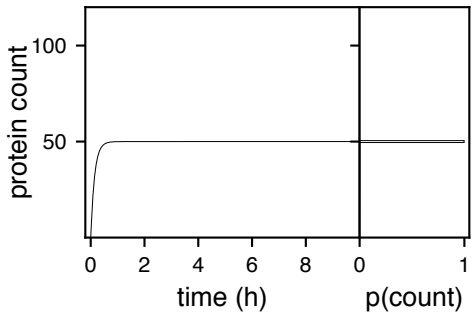


Figure 1.6: Time series from a deterministic simulation of the simple expression system (see Eq 1.2) with $k_1 = .1, k_2 = .1, k_3 = .1, k_4 = 2 \cdot 10^{-3}$. The subplot along the left side shows the epigenetic landscape (as calculated via a histogram of the adjacent time series).

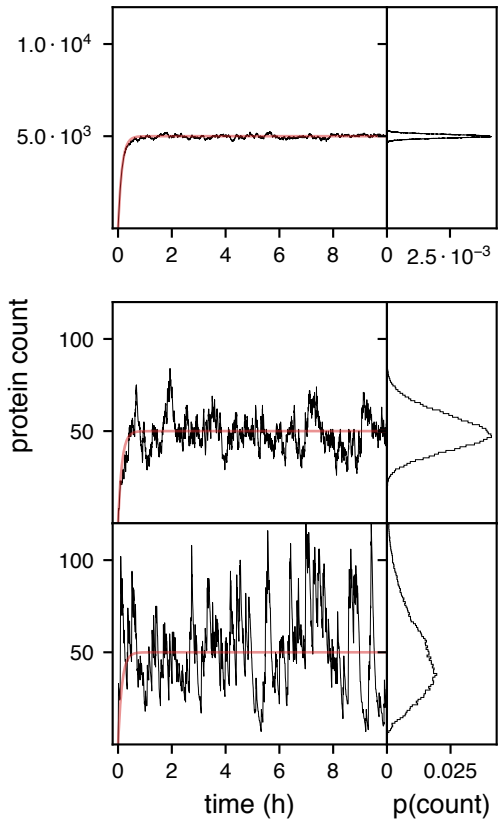


Figure 1.7: (Black lines) time series from stochastic simulations of the simple expression system (see Eq 1.2). The subplots along the left side show the epigenetic landscape of each variant (as calculated via a histogram of data from 10 simulation days, including the adjacent stochastic time series). (Red lines) the deterministic simulation of the same system, for comparison purposes. (Top panel) simple expression system with $k_1 = 10, k_2 = .1, k_3 = .1, k_4 = 2 \cdot 10^{-3}$ (Middle panel) same, with $k_1 = .1, k_2 = .1, k_3 = .1, k_4 = 2 \cdot 10^{-3}$. (Bottom panel) same, with $k_1 = .01, k_2 = 1, k_3 = .1, k_4 = 2 \cdot 10^{-3}$. The same constants were used in the deterministic and stochastic simulations (*i.e.* $c_i = k_i$). The mean expression level of the system simulated in the top panel is 100x that of the systems in the bottom two panels.

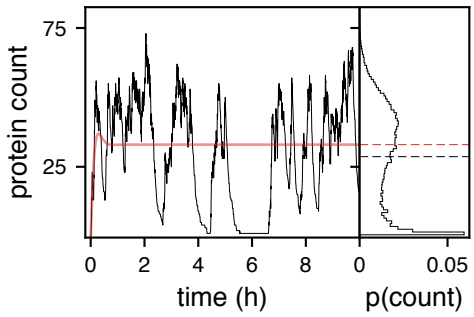


Figure 1.8: (Black line) time series from a stochastic simulation of the self regulating expression system (see Eq 1.3). (Red line) the deterministic simulation of the same system. The subplot along the left side shows the epigenetic landscape. (Black and red dashed lines) the mean protein expression value as calculated by stochastic and deterministic simulation, at 28.7 and 33.0 respectively (15% divergence). The deterministic rate constants used were $k_1 = .1, k_2 = .1, k_3 = .1, k_4 = 2 \cdot 10^{-3}, k_5 = 10^{-6}, k_6 = 1, k_7 = 1, k_8 = 2 \cdot 10^{-3}$. The stochastic rate constants were the same (*i.e.* $c_i = k_i$), except for $c_5 = 2 \cdot k_5 = 2 \cdot 10^{-6}$ (as per Eq 1.14).

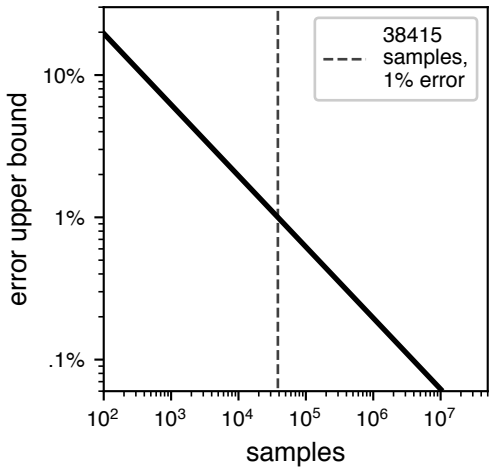


Figure 1.9: The margin of error (95% confidence) vs sample count when using stochastic simulation to calculate the $MFPT$ of a rare state switching event. The x-axis begins at 100 in order to emphasize that the relationship is only valid in the limit of large sample size. See Sec 2.1.4 for derivation and details.

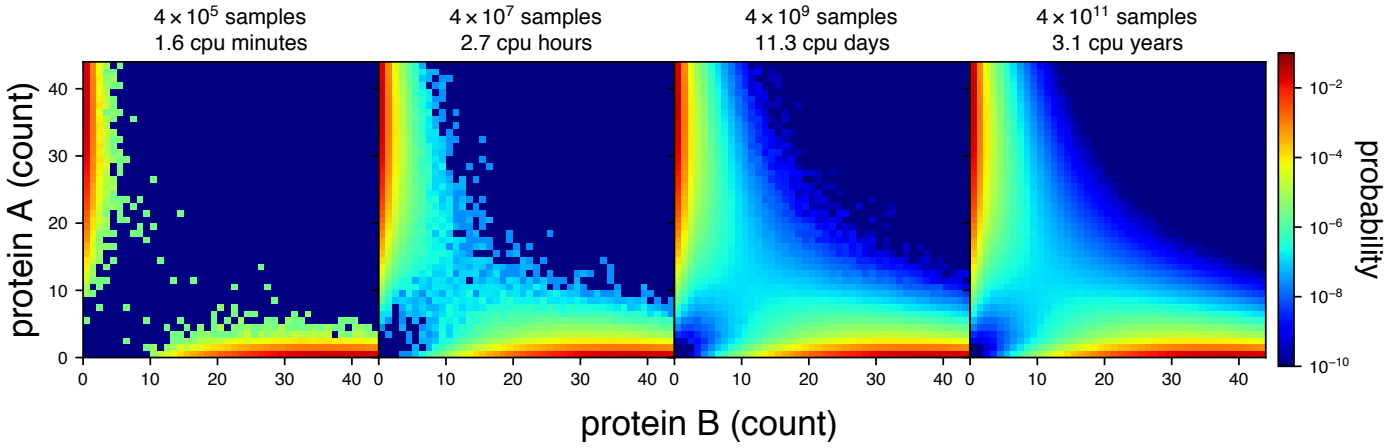


Figure 1.10: Two dimensional epigenetic landscapes generated from the $\text{GTS}_{\theta=.1}$ system (described in Sec 2.2.5). From left to right, the subplots show versions of the same landscape calculated using an increasing number of species count samples. The total sample count used in each landscape, and the CPU time used to collect those samples, is shown above each subplot. The samples were taken from 20 trajectories, each of which was generated using several thousand CPU cores running a parallelized implementation of SSA[85].

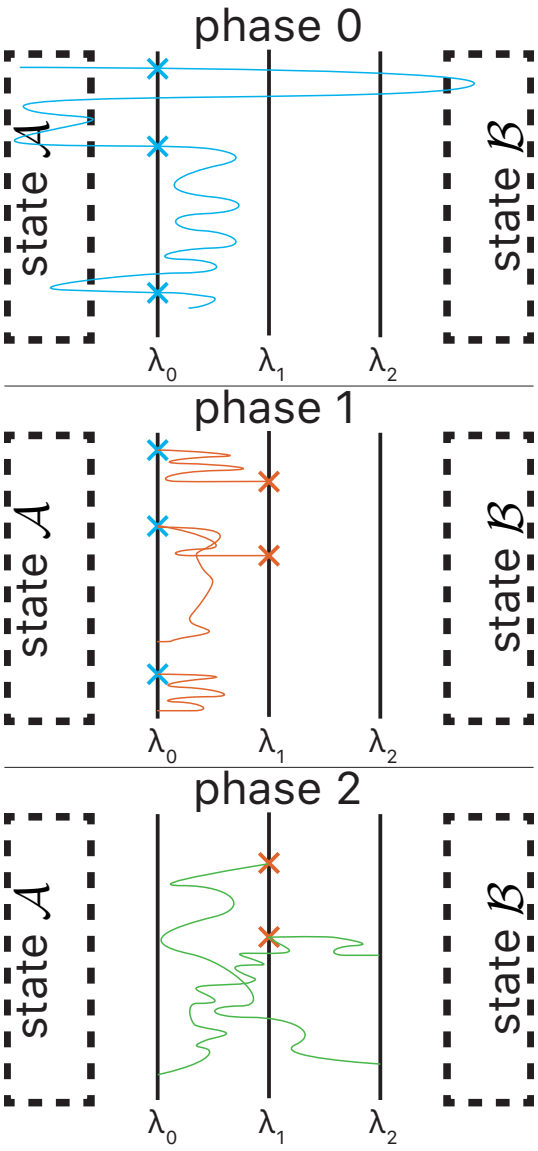


Figure 1.11: A schematic representation of an FFS simulation run from $\mathcal{A} \rightarrow \mathcal{B}$. The simulation shown has 3 phases, one for each interface λ_i that is defined.During each phase $i > 0$, trajectories are initialized from one of the points (chosen at random) where a trajectory in the previous phase crossed λ_{i-1} while traveling in the forward direction (*i.e.* towards $\mathcal{B}$). During phase 2 (the final phase), trajectories that cross λ_2 are considered to have completed the transition into $\mathcal{B}$.

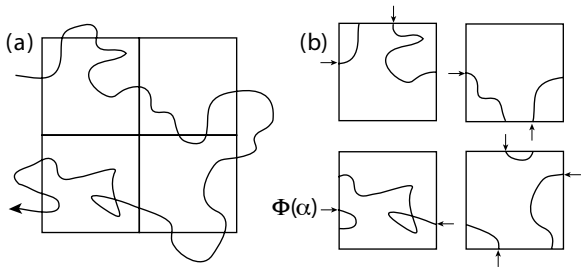


Figure 1.12: A simple schematic of a nonequilibrium umbrella sampling (NEUS) simulation. A state space divided into 4 regions along a 2D order parameter is shown. **a)** An example of a single unconstrained stochastic trajectory run over the regions and the surrounding area. **b)** An example of several NEUS trajectories constrained in each of the regions. When a trajectory leaves its starting region it is terminated, and a new trajectory is initialized at a state α within the same region. The choice of α is determined by the distribution of fluxes Φ_α into each α. Reprinted from Dickson et al., 2004[86].

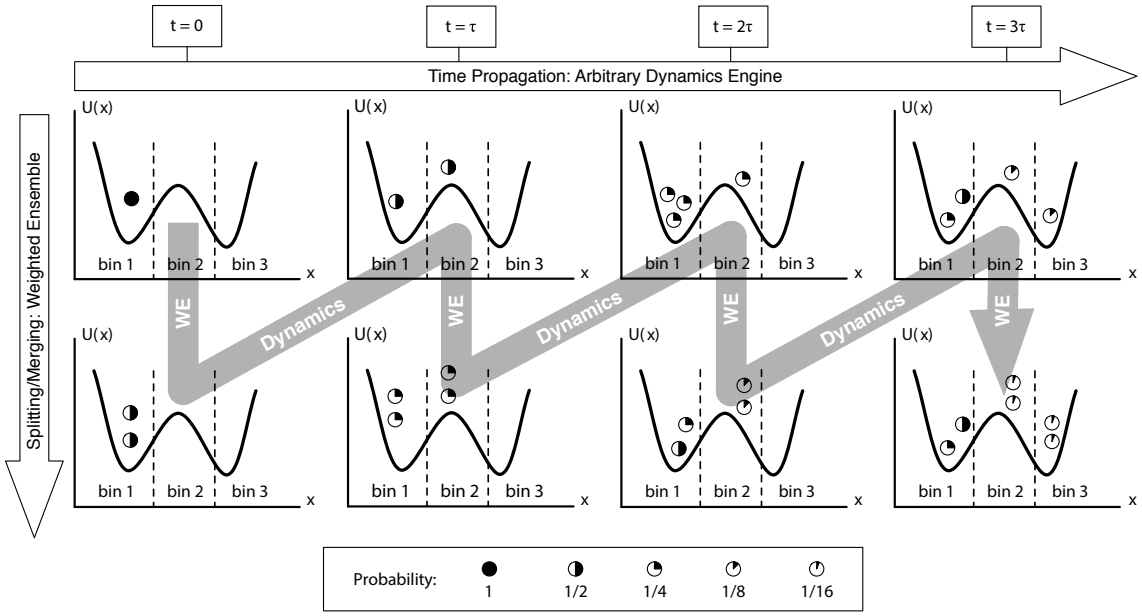


Figure 1.13: An explainer schematic demonstrating the inner workings of weighted ensemble (WE) simulation. The above WE kicks off by initializing a target count, $n = 2$, of unconstrained trajectories in the starting bin. Each trajectory is assigned an equal weight, and the weight all together sum to 1. After a certain time τ has passed, the trajectories are paused and the occupied bins are checked. If a bin has only 1 trajectory, it is split into two identical "daughters", each with half the weight. While there are more than 2 trajectories in a bin, they will be "merged" by terminating one and then adding its weight to one of the remaining trajectories. This cycle of unconstrained simulation followed by reweighting is then repeated iteratively. Reprinted from Donovan et al., 2013[74].

References

[1] Richard Rhodes. *The making of the atomic bomb*. New York: Simon & Schuster, 1986.

[2] W Johannsen. “The Genotype Conception of Heredity”. In: *The American Naturalist* 45.531 (1911), pp. 129–159.

[3] H Hellman. *Great Feuds in Medicine: Ten of the Liveliest Disputes Ever*. Medical Humanities collection PAH. Wiley, 2001.

[4] Francis Crick. “On protein synthesis.” In: *Symp. Soc. Exp. Biol.* 12 (1958), pp. 138–163.

[5] Francis Crick. “Central dogma of molecular biology”. In: *Nature* 227.5258 (1970), pp. 561–563.

[6] Conrad Hal Waddington. *Principles of Embryology*. 1965.

[7] Conrad Hall Waddington. *The Strategy of the Genes : a Discussion of Some Aspects of Theoretical Biology*. London: Allen and Unwin, 1957.

[8] Cyrus Levinthal. “How to Fold Graciously”. In: *Mossbauer Spectroscopy in Biological Systems: Proceedings of a meeting held at Allerton House, Monticello, Illinois*. Ed. by JTP Debrunnder and E Munck. University of Illinois Press, 1969, pp. 22–24.

[9] Li Xu, Xiakun Chu, Zhiqiang Yan, Xiliang Zheng, Kun Zhang, Feng Zhang, Han Yan, Wei Wu, and Jin Wang. “Uncovering the underlying physical mechanisms of biological systems via quantification of landscape and flux”. In: *Chinese Phys. B* 25.1 (2016), pp. 016401–27.

[10] Wei Wu and Jin Wang. “Potential and flux field landscape theory. II. Non-equilibrium thermodynamics of spatially inhomogeneous stochastic dynamical systems”. In: *J Chem Phys* 141.10 (2014), pp. 105104–48.

[11] Wei Wu and Jin Wang. “Potential and flux field landscape theory. I. Global stability and dynamics of spatially dependent non-equilibrium systems”. In: *J Chem Phys* 139.12 (2013), p. 121920.

[12] Chunhe Li and Jin Wang. “Landscape and flux reveal a new global view and physical quantification of mammalian cell cycle.” In: *Proc Natl Acad Sci USA* 111.39 (2014), pp. 14130–14135.

[13] Xiaosheng Luo, Liufang Xu, Bo Han, and Jin Wang. “Funneled potential and flux landscapes dictate the stabilities of both the states and the flow: Fission yeast cell cycle”. In: *PLoS Comput Biol* 13.9 (2017), e1005710–31.

[14] Chunhe Li and Jin Wang. “Quantifying the underlying landscape and paths of cancer.” In: *J R Soc Interface* 11.100 (2014), pp. 20140774–20140774.

[15] Armando Aranda-Anzaldo and Myrna A R Dent. “Landscaping the epigenetic landscape of cancer”. In: *Progress in Biophysics and Molecular Biology* (2018), pp. 1–20.

[16] Carl Adam Petri. “Communication with automata”. PhD thesis. University of Hamburg, 1966.

[17] P J Goss and J Peccoud. “Quantitative modeling of stochastic systems in molecular biology by using stochastic Petri nets.” In: *Proc Natl Acad Sci USA* 95.12 (1998), pp. 6750–6755.

[18] J W Pinney, D R Westhead, and G A McConkey. “Petri Net representations in systems biology.” In: *Biochem. Soc. Trans.* 31.Pt 6 (2003), pp. 1513–1515.

[19] Simon Hardy and Pierre N Robillard. “Modeling and simulation of molecular biology systems using petri nets: modeling goals of various approaches.” In: *J Bioinform Comput Biol* 2.4 (2004), pp. 595–613.

[20] Peter J Haas. *Stochastic Petri Nets*. Modelling, Stability, Simulation. Springer Science & Business Media, 2006.

[21] Darren J Wilkinson. *Stochastic Modelling for Systems Biology, Second Edition*. CRC Press, 2012.

[22] P Waage and C M Gulberg. “Studies concerning affinity”. In: *J. Chem. Educ.* 63.12 (1986), p. 1044.

[23] Wolfram Research, Inc. “Mathematica 11.0”. In: ().

[24] Alfred J Lotka. “Contribution to the Theory of Periodic Reactions”. In: *J. Phys. Chem. …* 14.3 (1909), pp. 271–274.

[25] Alfred J Lotka. *Elements of Physical Biology*. Williams and Wilkins, 1925.

[26] Loots I Dublin and Alfred J Lotka. “On the True Rate of Natural Increase: As Exemplified by the Population of the United States, 1920”. In: *Journal of the American Statistical Association* 20.151 (1925), pp. 305–339.

[27] Vito Volterra. “Fluctuations in the Abundance of a Species considered Mathematically”. In: *Nature* 118.2972 (1926), pp. 558–560.

[28] A M Turing. “The Chemical Basis of Morphogenesis”. In: *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences* 237.641 (1952), pp. 37–72.

[29] D T Gillespie. “Stochastic simulation of chemical kinetics”. In: *Annu Rev Phys Chem* (2007).

[30] N G Van Kampen. *Stochastic Processes in Physics and Chemistry*. Elsevier, 2011.

[31] J Goutsias and G Jenkinson. “Physics Reports”. In: *Physics Reports* 529.2 (2013), pp. 199–264.

[32] Elijah Roberts, Shay Be’er, Chris Bohrer, Rati Sharma, and Michael Assaf. “Dynamics of simple gene-network motifs subject to extrinsic fluctuations”. In: *Phys Rev E* 92.6 (2015), pp. 062717–14.

[33] D T Gillespie. “A general method for numerically simulating the stochastic time evolution of coupled chemical reactions”. In: *J Comput Phys* 22.4 (1976), pp. 403–434.

[34] Donald A McQuarrie. “Stochastic approach to chemical kinetics”. In: *Journal of Applied Probability* 4.3 (1967), pp. 413–478.

[35] I Oppenheim, K E Shuler, and G H Weiss. “Stochastic and Deterministic Formulation of Chemical Rate Equations”. In: *J Chem Phys* 50.1 (1969), pp. 460–466.

[36] Thomas G Kurtz. “The Relationship between Stochastic and Deterministic Models for Chemical Reactions”. In: *J Chem Phys* 57.7 (1972), pp. 2976–2978.

[37] D T Gillespie. “A rigorous derivation of the chemical master equation”. In: *Physica a-Statistical Mechanics and Its Applications* 188.1-3 (1992), pp. 404–425.

[38] Daniel T Gillespie. *Markov processes : an introduction for physical scientists*. Boston: Academic Press, 1992.

[39] Daniel T Gillespie, Andreas Hellander, and Linda R Petzold. “Perspective: Stochastic algorithms for chemical kinetics”. In: *J Chem Phys* 138.17 (2013), pp. 170901–15.

[40] Michael A Gibson and Jehoshua Bruck. “Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels”. In: *J Phys Chem A* 104.9 (2000), pp. 1876–1889.

[41] Yang Cao, Hong Li, and Linda Petzold. “Efficient formulation of the stochastic simulation algorithm for chemically reacting systems”. In: *J Chem Phys* 121.9 (2004), pp. 4059–4067.

[42] Larry Lok and Roger Brent. “Automatic generation of cellular reaction networks with Moleculizer 1.0”. In: *Nat Biotechnol* 23.1 (2005), pp. 131–136.

[43] James M McCollum, Gregory D Peterson, Chris D Cox, Michael L Simpson, and Nagiza F Samatova. “The sorting direct method for stochastic simulation of biochemical systems with varying reaction execution behavior”. In: *Computational Biology and Chemistry* 30.1 (2006), pp. 39–49.

[44] M B Elowitz. “Stochastic Gene Expression in a Single Cell”. In: *Science* 297.5584 (2002), pp. 1183–1186.

[45] Joanna Jaruszewicz and Tomasz Lipniacki. “Toggle switch: noise determines the winning gene”. In: *Phys Biol* 10.3 (2013), p. 035007.

[46] R Ahrends, A Ota, K M Kovary, T Kudo, B O Park, and M N Teruel. “Controlling low rates of cell differentiation through noise and ultrahigh feedback”. In: *Science* 344.6190 (2014), pp. 1384–1389.

[47] Burton W Andrews and Pablo A Iglesias. “An information-theoretic characterization of the optimal gradient sensing response of cells.” In: *PLoS Comput Biol* 3.8 (2007), e153.

[48] Gábor G Balázsi, Alexander A van Oudenaarden, and James J JJ Collins. “Cellular Decision Making and Biological Noise: From Microbes to Mammals”. In: *Cell* 144.6 (2011), pp. 910–925.

[49] Elisabet Pujadas and Andrew P Feinberg. “Regulated Noise in the Epigenetic Landscape of Development and Disease”. In: *Cell* 148.6 (2012), pp. 1123–1131.

[50] Sayuri K Hahl and Andreas Kremling. “A Comparison of Deterministic and Stochastic Modeling Approaches for Biochemical Reaction Systems: On Fixed Points, Means, and Modes”. In: *Front. Genet.* 7.46 (2016), p. 054103.

[51] Peters Baron. *Reaction Rate Theory and Rare Events*. First edition. Elsevier B.V., 2017.

[52] L F Shampine and C W Gear. “A User’s View of Solving Stiff Ordinary Differential Equations”. In: *SIAM Rev.* 21.1 (1979), pp. 1–17.

[53] Linda Petzold. “Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations”. In: *SIAM J. Sci. and Stat. Comput.* 4.1 (1983), pp. 136–148.

[54] David J Aldous. “Markov chains with almost exponential hitting times”. In: *Stochastic Processes and their Applications* 13.3 (1982), pp. 305–310.

[55] Herman Kahn and Theodore E Harris. “Estimation of particle transmission by random sampling”. In: *Nat Bur Stand Math* 12 (1951), pp. 27–30.

[56] John von Neumann. “Various Techniques Used in Connection with Random Digits”. In: *Nat Bur Stand Math* 12 (1951), pp. 36–38.

[57] H Kahn and A W Marshall. “Methods of Reducing Sample Size in Monte Carlo Computations”. In: *OR* 1.5 (1953), pp. 263–278.

[58] George E Forsythe. “Von Neumann’s Comparison Method for Random Sampling from the Normal and Other Distributions”. In: *Mathematics of Computation* 26.120 (1972), pp. 817–11.

[59] G A Huber and S Kim. “Weighted-ensemble Brownian dynamics simulations for protein association reactions.” In: *Biophys J* 70.1 (1996), pp. 97–110.

[60] Christoph Dellago, Peter G Bolhuis, Félix S Csajka, and David Chandler. “Transition path sampling and the calculation of rate constants”. In: *J Chem Phys* 108.5 (1998), pp. 1964–15.

[61] Titus S van Erp and Peter G Bolhuis. “Elaborating transition interface sampling methods”. In: *J Comput Phys* 205.1 (2005), pp. 157–181.

[62] Rafael C Bernardi, Marcelo C R Melo, and Klaus Schulten. “Enhanced sampling techniques in molecular dynamics simulations of biological systems”. In: *BBA - General Subjects* 1850.5 (2015), pp. 872–877.

[63] Rosalind J Allen, Patrick B Warren, and Pieter Rein ten Wolde. “Sampling rare switching events in biochemical networks.” In: *Phys Rev Lett* 94.1 (2005), p. 018104.

[64] Chantal Valeriani, Rosalind J Allen, Marco J Morelli, Daan Frenkel, and Pieter Rein ten Wolde. “Computing stationary distributions in equilibrium and nonequilibrium systems with forward flux sampling.” In: *J Chem Phys* 127.11 (2007), p. 114109.

[65] Rosalind J Allen, Daan Frenkel, and Pieter Rein ten Wolde. “Simulating rare events in equilibrium or nonequilibrium stochastic systems.” In: *J Chem Phys* 124.2 (2006), p. 024102.

[66] Rosalind J Allen, Chantal Valeriani, and Pieter Rein ten Wolde. “Forward flux sampling for rare event simulations.” In: *J Phys Condens Matter* 21.46 (2009), p. 463102.

[67] Aryeh Warmflash, Prabhakar Bhimalapuram, and Aaron R Dinner. “Umbrella sampling for nonequilibrium processes.” In: *J Chem Phys* 127.15 (2007), p. 154112.

[68] Eric Vanden-Eijnden and Maddalena Venturoli. “Exact rate calculations by trajectory parallelization and tilting”. In: *J Chem Phys* 131.4 (2009), p. 044120.

[69] Alex Dickson and Aaron R Dinner. “Enhanced sampling of nonequilibrium steady states.” In: *Annu Rev Phys Chem* 61.1 (2010), pp. 441–459.

[70] Alex Dickson, Aryeh Warmflash, and Aaron R Dinner. “Separating forward and backward pathways in nonequilibrium umbrella sampling”. In: *J Chem Phys* 131.15 (2009), pp. 154104–11.

[71] Daniel M Zuckerman and Lillian T Chong. “Weighted Ensemble Simulation: Review of Methodology, Applications, and Software”. In: *Annu Rev Biophys* 46.1 (2017), pp. 43–57.

[72] Lillian T Chong, Ali S Saglam, and Daniel M Zuckerman. “Path-sampling strategies for simulating rare events in biomolecular systems”. In: *Curr Opin Struct Biol* 43 (2017), pp. 88–94.

[73] Divesh Bhatt, Bin W Zhang, and Daniel M Zuckerman. “Steady-state simulations using weighted ensemble path sampling.” In: *J Chem Phys* 133.1 (2010), p. 014110.

[74] Rory M Donovan, Andrew J Sedgewick, James R Faeder, and Daniel M Zuckerman. “Efficient stochastic simulation of chemical kinetics networks using a weighted ensemble of trajectories.” In: *J Chem Phys* 139.11 (2013), p. 115105.

[75] Bin W Zhang, David Jasnow, and Daniel M Zuckerman. “The “weighted ensemble” path sampling method is statistically exact for a broad class of stochastic processes and binning procedures”. In: *J Chem Phys* 132.5 (2010), pp. 054107–8.

[76] Joshua L Adelman and Michael Grabe. “Simulating rare events using a weighted ensemble-based string method”. In: *J Chem Phys* 138.4 (2013), pp. 044105–14.

[77] Haoyun Feng, Ronan Costaouec, Eric Darve, and Jesús A Izaguirre. “A comparison of weighted ensemble and Markov state model methodologies”. In: *J Chem Phys* 142.21 (2015), pp. 214113–13.

[78] Jeremy O B Tempkin, Brian Van Koten, Jonathan C Mattingly, Aaron R Dinner, and Jonathan Weare. “Trajectory stratification of stochastic dynamics”. In: *arXiv* (2016). arXiv: 1610.09426v1 [cond-mat.stat-mech].

[79] H C Lie and J Quer. “Some connections between importance sampling and enhanced sampling methods in molecular dynamics”. In: *J Chem Phys* 147.19 (2017), p. 194107.

[80] Kai Kratzer, Axel Arnold, and Rosalind J Allen. “Automatic, optimized interface placement in forward flux sampling simulations”. In: *J Chem Phys* 138.16 (2013), pp. 164112–164112.

[81] Ao Ma and Aaron R Dinner. “Automatic Method for Identifying Reaction Coordinates in Complex Systems †”. In: *J Phys Chem B* 109.14 (2005), pp. 6769–6779.

[82] Ernesto E Borrero and Fernando A Escobedo. “Reaction coordinates and transition pathways of rare events via forward flux sampling”. In: *J Chem Phys* 127.16 (2007), p. 164101.

[83] Ernesto E Borrero and Fernando A Escobedo. “Optimizing the sampling and staging for simulations of rare events via forward flux sampling schemes”. In: *J Chem Phys* 129.2 (2008), pp. 024115–024117.

[84] Rosalind J Allen, Daan Frenkel, and Pieter Rein ten Wolde. “Forward flux sampling-type schemes for simulating rare events: efficiency analysis.” In: *J Chem Phys* 124.19 (2006), p. 194111.

[85] Elijah Roberts, John E Stone, and Zaida Luthey-Schulten. “Lattice Microbes: high-performance stochastic simulation method for the reaction-diffusion master equation.” In: *J Comput Chem* 34.3 (2013), pp. 245–255.

[86] Alex Dickson, Aryeh Warmflash, and Aaron R Dinner. “Nonequilibrium umbrella sampling in spaces of many order parameters.” In: *J Chem Phys* 130.7 (2009), p. 074104.

Chapter 2

Automatic error control during forward flux sampling of rare events in master equation models

2.1 Theory and methods

2.1.1 Simulation of rare events in stochastic processes

The standard stochastic simulation protocol, here referred to as direct sampling (DS), proceeds in two iterated steps: 1) increment the system time with the time of the next stochastic event, 2) update the system state according to the event. These two steps are repeated until some predetermined stopping condition (simulation steps, simulation time, etc.) is reached, at which point the simulation is terminated. Repeated DS simulation produces a data set from which ensemble average quantities can be estimated by straightforward averaging. Running more DS replicates leads to increased accuracy.

As discussed above, DS has the disadvantage that it requires a long simulation time to sample each rare event. Therefore, calculating rare event statistics is computationally expensive. Enhanced sampling (ES) methods use a combination of constraints on simulation trajectories and statistical unbiasing methods in order to enrich the sampling of rare events. Forward flux sampling (FFS) is a popular ES method that was initially proposed by Allen and coworkers[1]. We implemented the

FFS algorithm as follows:

1. Choose two points of interest in the state space of the model system, $\mathcal{A}$ and $\mathcal{B}$. These will serve as an initial and a final state for the simulation.

2. Choose a one dimensional parameter $\mathcal{O}$ for which $\mathcal{O}(\mathcal{A}) \neq \mathcal{O}(\mathcal{B})$. If $\mathcal{O}(\mathcal{A}) > \mathcal{O}(\mathcal{B})$, swap $\mathcal{A} \Leftrightarrow \mathcal{B}$.

3. Choose a set $\{\lambda_0, \cdots, \lambda_N\}$ of interface values of $\mathcal{O}$ such that $\mathcal{O}(\mathcal{A}) < \lambda_i < \mathcal{O}(\mathcal{B})$ for every λ_i. We say that a trajectory has fluxed forward with respect to a λ_i when it crosses the interface traveling in the direction of increasing $\mathcal{O}$, and that it has fluxed backward when it crosses traveling in the direction of decreasing $\mathcal{O}$.

4. Begin FFS phase 0:

   (a) Execute a DS trajectory initialized at $\mathcal{A}$. If the trajectory fluxes forward across λ_0, record the elapsed simulation time τ since the previous crossing event, and also record the state $\mathcal{X}$. Multiple phase 0 trajectories can be executed in parallel.

   (b) If the trajectory ever crosses the final interface λ_N, pause the tracking of the waiting time until the trajectory moves backward across λ_0.

   (c) Once n_0 samples of τ have been collected, terminate the trajectories and move to the next phase.

5. Begin FFS phase $i > 0$:

   (a) Randomly choose a state $\mathcal{X}$ from the collection of states at which any trajectory in the previous phase crossed λ_{i-1}.

   (b) Execute a DS trajectory initialized at $\mathcal{X}$. Allow the trajectory to run until it either crosses λ_0 back into $\mathcal{A}$ or moves forward across λ_i. Terminate the

trajectory and record a trajectory outcome value, either 0 or 1, depending on whether the trajectory moved backward or forward, respectively. If the trajectory moved forward, add the endpoint, which will lie along λ_i, to the set of states that will be used to initialize trajectories during the next phase.

(c) Repeat steps 5a-b.

(d) Once n_i trajectory outcomes have been collected terminate the phase. Every phase i trajectory can be executed in parallel.

6. The procedure for FFS phase i is then repeated for phase $i+1$ (during which trajectories are launched from λ_i) until the final phase, phase N, is reached. Trajectories in phase N begin at λ_{N-1} and may move forward across λ_N and into $\mathcal{B}$.

The overall aim of FFS is to ratchet a simulation from $\mathcal{A} \to \mathcal{B}$ across state space. The immediate goal of each phase is to estimate a phase weight. During phase i, samples are taken from an observable $\mathbb{w}_i$ that behaves as a random variable. $\mathbb{w}_0$ is the waiting time in between forward flux events across λ_0, and each $\mathbb{w}_{i>0}$ is the final outcome (0 for fall back or 1 for flux forward) of each trajectory launched in that i. The phase weight w_i is defined as the true expected value of the random variable $\mathbb{w}_i$:

$$w_i = E\left[\mathbb{w}_i\right] = \begin{cases} \tau_{\mathcal{A}} & \text{if } i = 0, \\ P\left(\lambda_i \mid \lambda_{i-1}\right) & \text{otherwise,} \end{cases} \tag{2.1}$$

where $\tau_{\mathcal{A}}$ is the expected waiting time in between λ_0 crossing events, and $P\left(\lambda_i \mid \lambda_{i-1}\right)$ is the probability that a trajectory launched from λ_{i-1} (*i.e.* launched during phase i) crosses forward past λ_i before it falls back behind the starting interface λ_0. The phase weights can be used to reweight the output of an FFS simulation so as to calculate unbiased statistics.

2.1.2 Estimating values of interest from DS and ES stochastic simulations

As mentioned in the previous section, estimating values of interest from DS simulation is straightforward. One valid estimator of any ensemble average quantity is simply the arithmetic mean taken across an appropriate set of direct observations. For example, in order to calculate the $MFPT$ of the switching process that takes some multistate system from $\mathcal{A} \to \mathcal{B}$, n DS simulations are initialized in $\mathcal{A}$. Each of these n replicate simulations is allowed to run until the first time it enters $\mathcal{B}$, at which point it is terminated. The time that each simulation i ran for is then a single sample FPT_i of the first passage time of the switching process. The mean of these first passage time samples is an estimate of $MFPT$[2]:

$$\widehat{MFPT}_{\text{ds}} = \frac{1}{n}\sum_{i=1}^{n} FPT_i, \tag{2.2}$$

where $\widehat{MFPT}_{\text{ds}}$ is the estimator (*i.e.* estimation function) of $MFPT$ specific to DS simulation. Throughout this paper, we place a $\widehat{\ }$ above a symbol to indicate that we are referring to an estimator of a value rather than to the value itself.

Equivalent ensemble average estimators can be calculated using FFS. Although the results of FFS simulations are biased and partitioned, statistical protocols allow FFS results to be recombined and reweighted so as to recapitulate the results of the unbiased ensemble. In order to perform these reweightings, we need estimates of the phase weights. During phase i of FFS, n_i samples $\left\{\mathbb{w}_{i1}\,,\ \cdots\,,\ \mathbb{w}_{in_i}\right\}$ of an underlying random process $\mathbb{w}_i$ are taken. The mean of this sample can be used as an estimator $\widehat{w}_i$ of phase weight i:

$$\widehat{w}_i = \frac{\sum_{j=1}^{n_i} \mathbb{w}_{ij}}{n_i} = \begin{cases} \frac{\sum_{j=1}^{n_0} \tau_i}{n_0} & \text{if } i = 0, \\ \frac{n_i^s}{n_i} & \text{otherwise,} \end{cases} \tag{2.3}$$

where each τ_i is a sample of the waiting time in between phase 0 forward flux events and n_i^s is the total count of trajectories that successfully fluxed forward during phase $i > 0$.

In an idealized situation in which it were possible to know the exact values of the phase weights w_i, the exact value of the $MFPT$ could be found via a simple combination W of the phase weights:

$$MFPT = \frac{\tau_{\mathcal{A}}}{\prod_{i>0} P\left(\lambda_i \mid \lambda_{i-1}\right)} = \frac{w_0}{\prod_{i>0} w_i} = W. \tag{2.4}$$

In reality, we only know the phase weight estimators $\widehat{w}_i$, so we must instead estimate the value of the $MFPT$ by way of the combination of estimators $\widehat{W}$:

$$\widehat{MFPT}_{\text{ffs}} = \frac{\widehat{w}_0}{\prod_{i>0} \widehat{w}_i} = \widehat{W}. \tag{2.5}$$

where $\widehat{MFPT}_{\text{ffs}}$ is the estimator of $MFPT$ specific to FFS simulation. Other quantities of interest, such as the stationary PDF, can also be calculated using the phase weights[3].

2.1.3 Predicting simulation error in terms of margin of error

Stochastic simulation results are not single valued, but are instead estimators, random variables with associated distributions. For example, when attempting to calculate $MFPT$, each complete round of simulations can be thought of as performing a single draw from the distribution of possible $\widehat{MFPT}$ values. A prediction about the likely level of error in any given set of simulations can be made based on the characteristics of this estimate distribution. One way to quantify confidence in this prediction is in terms of the margin of error. The margin of error of a random variable X is defined as the ratio of half the width of a specified confidence interval and its expected value:

$$\zeta\left[X, \alpha\right] = \frac{ub_\alpha\left[X\right] - lb_\alpha\left[X\right]}{2E\left[X\right]}, \tag{2.6}$$

where $\zeta\left[X, \alpha\right]$ is the margin of error of X at confidence level α, $E\left[X\right]$ is the expected value of X, and $lb_\alpha\left[X\right]$ and $ub_\alpha\left[X\right]$ are the lower and upper confidence bounds, respectively.

For many values of interest $\zeta[X, \alpha]$ is dependent on simulation parameters that are set by the user. For example, when calculating $MFPT$, the margin of error $\zeta\left[\widehat{MFPT}, \alpha\right]$ is dependent upon the total number of replicate simulations (when using DS), or upon the count of trajectories run in each phase (when using FFS).

2.1.4 Determining margin of error from simulation parameters

It is straightforward to determine the margin of error of an estimator that depends on a single underlying observable. For example, consider the margin of error of an $MFPT$ estimate as determined via DS simulation, $\zeta\left[\widehat{MFPT}_{\text{ds}}, \alpha\right]$. It has been shown that observations of first passage times between two well separated states follow an exponential distribution during DS simulations[4]. Given this distribution of first passage times, the central limit theorem[5] gives the distribution of $MFPT$ estimates in the limit of large sample size:

$$\frac{\sqrt{n}\left(\widehat{MFPT}_{\text{ds}} - E[FPT]\right)}{\sqrt{V[FPT]}} \xrightarrow{D} \mathcal{N}(0,1), \tag{2.7}$$

where n is the count of first passage time observations, $E[FPT] = MFPT$ is the expected value of the first passage time, $V[FPT]$ is the variance, $\mathcal{N}(0,1)$ is the standard normal distribution (*i.e.* mean 0 and variance 1), and $\xrightarrow{D}$ signifies that the distribution on the left converges to the one on the right. Eq 2.7 implies[5] that an estimate of $MFPT$ calculated using n observations will follow a normal distribution such that:

$$\begin{aligned} E\left[\widehat{MFPT}_{\text{ds}}\right] &= MFPT, \\ V\left[\widehat{MFPT}_{\text{ds}}\right] &= \frac{V[FPT]}{n} = \frac{MFPT^2}{n}, \end{aligned} \tag{2.8}$$

where the fact that $V = E^2$ for an exponential distribution was used to factor out $V[FPT]$. The lower and upper bounds of the confidence interval of any normally distributed random variable X can be found using a standard formula[6] that depends

on the first two moments of X:

$$lb_{\alpha}[X] = E[X] - z_{\alpha}\sqrt{V[X]},$$
$$ub_{\alpha}[X] = E[X] + z_{\alpha}\sqrt{V[X]}, \tag{2.9}$$

where z_{α} is the z score associated with the confidence level α (*e.g.* $z_{.95} \approx 1.96$)*. Plugging Eqs 2.8 and 2.9 into Eq 2.6 yields the margin of error of the DS $MFPT$ estimator:

$$\zeta\left[\widehat{MFPT}_{\mathsf{ds}}, \alpha\right] = \frac{z_{\alpha}}{\sqrt{n}}. \tag{2.10}$$

Thus, if say, 10^5 replicate trajectories are produced during a DS simulation, the resultant estimate of $MFPT$ will have no more than $\frac{z_{.95}}{\sqrt{10^5}} \approx \frac{1.96}{316} = 0.62\%$ difference from the true value 95% of the time.

2.1.5 Minimizing the computational cost required to achieve a desired error goal

We define the computational cost $\mathcal{C}$ of a simulation to be equivalent to the average in-simulation time required to complete it (alternatively, one may use the count of simulation steps). When estimating $MFPT$ using DS simulation, the computational cost is:

$$\mathcal{C}_{\mathsf{ds}} = n \cdot MFPT, \tag{2.11}$$

where n is the total number of replicate trajectories. Eq 2.10 illustrates the direct tradeoff between computational cost and simulation accuracy. In order to find the minimum number of replicate trajectories that are required to achieve a particular error goal in DS simulations, Eq 2.10 can be solved for n:

$$n = \left(\frac{z_{\alpha}}{\zeta}\right)^2 \tag{2.12}$$

*The z score z_{α} for any confidence level α can be calculated as $z_{\alpha} = \sqrt{2}\mathrm{Erfinv}(\alpha)$, where Erfinv is the Inverse Error Function[7], and α is expressed as a fraction. Some authors write the z score as $z_{(1-\alpha)/2}$ instead of z_{α}[6].

2.2 Results

2.2.1 Derivation of the FFPilot optimizing equation

In this subsection we find the choice of number trajectories to launch in each FFS phase (which we will also refer to as sample count or n_i) that will minimize simulation run time while fixing the margin of error ζ of the estimator of $MFPT$. We will refer to the $MFPT$ estimator in this subsection as $\widehat{W}$. The derivation of the optimal choice of n_i starts with the characterization of the moments and distribution of $\widehat{W}$. We then use the moments and distribution of $\widehat{W}$ to find ζ as a function of the per-phase sample counts n_i. We then use the method of Lagrange multipliers to find the optimal choice of n_i that will keep ζ fixed while minimizing simulation run time.

2.2.1.1 The moments and distribution of the FFS $MFPT$ estimator $\widehat{W}$

Each individual FFS phase weight w_i can be thought of as the first moment (*i.e.* the mean) of an observable $\mathbb{w}_i$ of a random process that is specific to phase i. By the end of every FFS phase i, n_i samples have been drawn from $\mathbb{w}_i$ (see Sec 2.1.1 for a more concrete description), at which point the phase weight w_i is estimated as:

$$\widehat{w}_i = \frac{1}{n_i} \sum_{j=1}^{n_i} \mathbb{w}_{ij}$$

Given the above form of $\widehat{w}_i$, and given that the observable $\mathbb{w}_i$ meets certain regularity conditions[8], the asymptotic distribution of the individual $\widehat{w}_i$ terms can be determined from the central limit theorem:

$$\frac{\sqrt{n_i}(\widehat{w}_i - w_i)}{\sqrt{V\left[\mathbb{w}_i\right]}} \xrightarrow{D} \mathcal{N}\left(0, 1\right). \tag{2.13}$$

The moments of each $\widehat{w}_i$ can be determined by the appropriate interpretation of Eq 2.13:

$$\begin{aligned} E\left[\widehat{w}_i\right] &= w_i, \\ V\left[\widehat{w}_i\right] &= \frac{V\left[\mathtt{w}_i\right]}{n_i}. \end{aligned} \tag{2.14}$$

An estimator with this type of convergence behavior is said to be $\sqrt{n}$-consistent.

Given that the $MFPT$ estimator $\widehat{W}$ is defined in Eq 2.5 as a function $g\left[\widehat{w}\right]$ of a vector of $\sqrt{n}$-consistent estimators $\widehat{w} = \{\widehat{w}_0\,,\ \widehat{w}_1\,,\ \cdots\,,\ \widehat{w}_N\}$, we can apply the multivariate delta method[9] to determine the moments and distribution of $\widehat{W}$. From the multivariate delta method, we know that:

$$\frac{\widehat{W} - w_0 \prod_{i>0} {w_i}^{-1}}{\sqrt{\nabla_{\widehat{w}}^T \Sigma \nabla_{\widehat{w}}}} \xrightarrow{D} \mathcal{N}\left(0, 1\right), \tag{2.15}$$

where ∇_w is the gradient of $g(\widehat{w})$ evaluated at w, ∇_w^T is the transpose of ∇_w, and Σ is the covariance matrix of the phase weight estimators $\widehat{w}_i$. We can determine the moments of $\widehat{W}$ by interpreting Eq 2.15:

$$\begin{aligned} E\left[\widehat{W}\right] &= \frac{w_0}{\prod_{i>0} w_i} = MFPT, \\ V\left[\widehat{W}\right] &= \nabla_w^T \Sigma \nabla_w. \end{aligned} \tag{2.16}$$

A simpler form of $V\left[\widehat{W}\right]$ can be found. To begin with, we find ∇_w. In column vector form it is:

$$\nabla_w = \begin{pmatrix} {w_0}^{-1} w_0 \prod_{j>0} {w_j}^{-1} \\ -{w_1}^{-1} w_0 \prod_{j>0} {w_j}^{-1} \\ \vdots \\ -{w_N}^{-1} w_0 \prod_{j>0} {w_j}^{-1} \end{pmatrix} = \begin{pmatrix} s_0 \\ -s_1 \\ \vdots \\ -s_N \end{pmatrix}, \tag{2.17}$$

where we have substituted s_i for ${w_i}^{-1} w_0 \prod_{j>0} {w_j}^{-1}$ for the sake of brevity. If the covariance of $\widehat{w}_i$ and $\widehat{w}_j$ is written as σ_{ij}, then the covariance matrix Σ is:

$$\Sigma = \begin{pmatrix} \sigma_{00} & \sigma_{01} & \cdots & \sigma_{0N} \\ \sigma_{10} & \sigma_{11} & \cdots & \sigma_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{N0} & \sigma_{N1} & \cdots & \sigma_{NN} \end{pmatrix},$$

and the variance of $\widehat{W}$ can be written as:

$$V\left[\widehat{W}\right] = \begin{pmatrix} s_0 & -s_1 & \cdots & -s_N \end{pmatrix} \begin{pmatrix} \sigma_{00} & \sigma_{01} & \cdots & \sigma_{0N} \\ \sigma_{10} & \sigma_{11} & \cdots & \sigma_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{N0} & \sigma_{N1} & \cdots & \sigma_{NN} \end{pmatrix} \begin{pmatrix} s_0 \\ -s_1 \\ \vdots \\ -s_N \end{pmatrix}. \tag{2.18}$$

If we impose the assumption of independence on all of the phase weight estimators $\widehat{w}_i$, then only the diagonal elements of the covariance are non-zero:

$$V\left[\widehat{W}\right] = \begin{pmatrix} s_0 & -s_1 & \cdots & -s_N \end{pmatrix} \begin{pmatrix} \sigma_{00} & 0 & \cdots & 0 \\ 0 & \sigma_{11} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_{NN} \end{pmatrix} \begin{pmatrix} s_0 \\ -s_1 \\ \vdots \\ -s_N \end{pmatrix}.$$

Under this condition of independence the form of $V\left[\widehat{W}\right]$ can be simplified considerably:

$$V\left[\widehat{W}\right] = \sum_i s_i^2 \sigma_{ii},$$

$$V\left[\widehat{W}\right] = \sum_i {w_i}^{-2} \sigma_{ii} \frac{{w_0}^2}{\prod_{j>0} {w_j}^2},$$

and since $\sigma_{ii} = V\left[\widehat{w}_i\right]$:

$$V\left[\widehat{W}\right] = \frac{{w_0}^2}{\prod_{j>0} {w_j}^2} \sum_i \frac{V\left[\widehat{w}_i\right]}{{w_i}^2}.$$

Finally, plugging in substitutions from Eq 2.4 and from Eq 2.14, the variance of $\widehat{W}$ is:

$$V\left[\widehat{W}\right] = MFPT^2 \sum_i \frac{V\left[\mathbb{w}_i\right]}{{w_i}^2 n_i}. \tag{2.19}$$

Alternatively, the variance of $\widehat{W}$ can be derived from the formula for the variance of a product of random variables (see supplemental Sec 2.S.1). The expressions derived from each technique agree in the high sample count limit.

The result in Eq 2.19 agrees with and is similar to the established result of Allen *et al.*[10] concerning the variance of estimates produced by a complete FFS simulation. Unlike earlier work, however, we have imposed no particular form on $\mathbb{w}_i$,

the random process underlying each phase i. As will be seen in Sec 2.2.1.4, this generalization allows us to study the contributions of phase 0 to the overall error of an FFS simulation for the first time.

2.2.1.2 Margin of error of $\widehat{W}$

Now we derive a formula for $\zeta\left[\widehat{W},\alpha\right]$, the margin of error of the $MFPT$ estimator $\widehat{W}$. From Eq 2.15 we know that $\widehat{W}$ follows a normal distribution. The lower and upper confidence bounds of $\widehat{W}$ are found by plugging the the moments of $\widehat{W}$ (given by Eqs 2.16 and 2.19) into the bounds formulas for a normally distributed random variable (given by Eq 2.9):

$$\begin{aligned} ub_\alpha\left[\widehat{W}\right] &= MFPT\left(1+z_\alpha\sqrt{\sum_i \frac{V\left[\mathbb{w}_i\right]}{{w_i}^2 n_i}}\right), \\ lb_\alpha\left[\widehat{W}\right] &= MFPT\left(1-z_\alpha\sqrt{\sum_i \frac{V\left[\mathbb{w}_i\right]}{{w_i}^2 n_i}}\right). \end{aligned} \tag{2.20}$$

Plugging Eq 2.20 into the margin of error definition (given by Eq 2.6) yields the desired margin of error:

$$\zeta\left[\widehat{W},\alpha\right] = z_\alpha\sqrt{\sum_i \frac{V\left[\mathbb{w}_i\right]}{{w_i}^2 n_i}}. \tag{2.21}$$

2.2.1.3 Derivation of the general optimizing equation

As we can see from Eq 2.21, there are many different choices of n_i that will give the same value of $\zeta\left[\widehat{W},\alpha\right]$. What we really want to find is the optimal choice of n_i that will minimize simulation run time while keeping $\zeta\left[\widehat{W},\alpha\right]$ fixed. We can find a formula for this optimal choice using the method of Lagrange multipliers[11].

For the method of Lagrange multipliers, we need a function to minimize, the target function $f[x]$, and a function to hold constant, the constraint equation $g[x]$. We use the total computational cost $\mathcal{C}$ as the target function, which for FFS is:

$$f\left[n_i\right] = \mathcal{C}_{\text{ffs}} = \sum_{i=0}^{N} n_i\, c_i \tag{2.22}$$

where c_i is the average computational cost per sample. For the constraint equation, we square both sides of Eq 2.21 and set it equal to zero:

$$g\left[n_i\right] = \sum_i \frac{k_i}{n_i} - \frac{\zeta^2}{{z_\alpha}^2} = 0, \tag{2.23}$$

where

$$k_i = \frac{V\left[\mathtt{w}_i\right]}{{w_i}^2}.$$

Now that we've chosen a target and a constraint function, the next step of the method is to combine Eqs 2.22 and 2.23 in order to write out the Lagrangian:

$$\mathcal{L} = \sum_i n_i \, c_i + \lambda \left(\sum_i \frac{k_i}{n_i} - \frac{\zeta^2}{{z_\alpha}^2} \right), \tag{2.24}$$

and then find its gradient $\nabla\mathcal{L}$:

$$\nabla\mathcal{L} = \left\{\partial_\lambda \mathcal{L}, \partial_{n_0}\mathcal{L}, ..., \partial_{n_N}\mathcal{L}\right\},$$

We find the values of each of the partial derivatives individually. Calculating $\partial_\lambda\mathcal{L}$ is trivial, and when finding each separate $\partial_{n_i}\mathcal{L}$ we can eliminate all but one term from both sums, since terms that don't depend on n_i will vanish:

$$\partial_\lambda \mathcal{L} = \sum_i \frac{k_i}{n_i} - \frac{\zeta^2}{{z_\alpha}^2}.$$

$$\partial_{n_i}\mathcal{L} = c_i - \frac{\lambda k_i}{n_i^2},$$

Now we can write out the actual gradient

$$\nabla\mathcal{L} = \left\{ \sum_i \frac{k_i}{n_i} - \frac{\zeta^2}{{z_\alpha}^2}, ..., c_i - \frac{\lambda k_i}{n_i^2}, ... \right\}. \tag{2.25}$$

The next step is to set each component of the gradient Eq 2.25 equal to zero and solve the resulting set of $N + 2$ equations for λ and each n_i. We begin by solving $\partial_{n_i}\mathcal{L} = 0$ for n_i in terms of λ:

$$c_i - \frac{\lambda k_i}{n_i^2} = 0,$$

$$n_i = \left\{ -\frac{\sqrt{\lambda}\sqrt{k_i}}{\sqrt{c_i}}, \frac{\sqrt{\lambda}\sqrt{k_i}}{\sqrt{c_i}} \right\}. \tag{2.26}$$

Next, we solve $\partial_\lambda \mathcal{L} = 0$ for $\sqrt{\lambda}$ by substituting in the positive expression for n_i found in Eq 2.26:

$$\sum_i \frac{k_i}{\frac{\sqrt{\lambda}\sqrt{k_i}}{\sqrt{c_i}}} - \frac{\zeta^2}{{z_\alpha}^2} = 0,$$

$$\sqrt{\lambda} = \frac{{z_\alpha}^2}{\zeta^2} \sum_i \sqrt{c_i\, k_i}. \tag{2.27}$$

Now we eliminate λ from our expression for n_i in Eq 2.26 by substituting in the expression for $\sqrt{\lambda}$ we found in Eq 2.27:

$$n_i = \frac{{z_\alpha}^2}{\zeta^2} \sqrt{\frac{k_i}{c_i}} \sum_j \sqrt{c_j\, k_j}. \tag{2.28}$$

2.2.1.4 The optimizing equation for FFS

A form of the general optimizing equation given in Eq 2.28 that is more specific to FFS can be found by considering the properties of the observable $\mathbb{w}_i$ in each phase. During a phase $i > 0$, each trajectory launched and finished is equivalent to a single sample taken from $\mathbb{w}_{i>0}$. The jth trajectory of a phase will either succeed (*i.e.* cross forward to the next interface) with probability p_{ij}, or it will fail (*i.e.* fall back into its starting basin), with probability $1-p_{ij}$. The sample taken from $\mathbb{w}_{i>0}$ is 1 If the trajectory succeeds, and 0 otherwise. Thus, the outcome of the jth trajectory of phase i is a Bernoulli random variable with probability parameter p_{ij}.

For multidimensional systems, p_{ij} is dependent upon the starting point of a trajectory. Since the starting point of each trajectory is chosen at random, p_{ij} in general varies from trajectory to trajectory. Thus, each $\mathbb{w}_{i>0}$ is technically a mixture of Bernoulli random variables. However, for the purposes of the optimizing equation, it can be shown that no accuracy is lost if each $\mathbb{w}_{i>0}$ is treated as a single Bernoulli

random variable (see Appendix 2.A.1) with moments:

$$E\left[\mathbb{w}_{i>0}\right] = w_i = p_i,$$
$$V\left[\mathbb{w}_{i>0}\right] = p_i\left(1-p_i\right), \tag{2.29}$$

where p_i is the crossing probability in phase i (*i.e.* $P\left(\lambda_i \mid \lambda_{i-1}\right)$).

The k_i terms in Eq 2.28 can be expanded to yield:

$$n_i = \frac{{z_\alpha}^2}{\zeta^2}\sqrt{\frac{V\left[\mathbb{w}_i\right]}{{w_i}^2\, c_i}}\sum_j \sqrt{\frac{V\left[\mathbb{w}_j\right]\, c_j}{{w_j}^2}}. \tag{2.30}$$

The moments from Eq 2.29 can then be plugged into Eq 2.30 to yield the FFS specific form of the optimizing equation:

$$n_i = \begin{cases} \frac{{z_\alpha}^2}{\zeta^2}\sqrt{\frac{V[\mathbb{w}_0]}{{w_0}^2 c_0}}\left(\sqrt{\frac{V[\mathbb{w}_0]c_0}{{w_0}^2}} + \sum_{j=1}^{N}\sqrt{\frac{(1-p_j)c_j}{p_j}}\right) & \text{if } i=0, \\ \frac{{z_\alpha}^2}{\zeta^2}\sqrt{\frac{1-p_i}{p_i c_i}}\left(\sqrt{\frac{V[\mathbb{w}_0]c_0}{{w_0}^2}} + \sum_{j=1}^{N}\sqrt{\frac{(1-p_j)c_j}{p_j}}\right) & \text{otherwise.} \end{cases} \tag{2.31}$$

We call this form the FFPilot optimizing equation.

In regards to phase 0, the precise forms of $E\left[\mathbb{w}_0\right] = {w_0}^2$ and $V\left[\mathbb{w}_0\right]$ are unknown. We have found that $\mathbb{w}_0$, the waiting time in between phase 0 forward flux events, does not in general follow an exponential distribution (see supplemental Fig S2). In fact, the distribution of $\mathbb{w}_0$ seems to be highly model dependent. Thus, in order to avoid any assumptions about $\mathbb{w}_0$, we leave the ratio $\frac{V[\mathbb{w}_0]}{{w_0}^2}$ unexpanded in Eq 2.31.

Of the assumptions made in deriving Eq 2.31, two of the most significant are the assumption of large sample size, and the assumption of the uncorrelatedness of the phases during an FFS simulation. The large sample size assumption underlies the validity of Eqs 2.14 and 2.15. In general this assumption can be satisfied by setting a minimum floor on the number of samples n_i taken in each phase (an implementation of this sample size floor is discussed in the next section).

Ensuring that the uncorrelatedness assumption is satisfied is altogether trickier. Most of the preexisting FFS literature takes the uncorrelatedness of the phases as a given[10,12,13], but in practice we have found this to not always be the case. This

issue is discussed in detail in Sec 2.2.6. In brief, systems with complex, high dimensional state spaces tend to have correlations between the outcomes of trajectories across the different phases. This results in non-zero covariance between the different phase weights, which effectively like adding an extra term to Eq 2.19. In other words, when the phases are correlated, our approach will somewhat underestimate the actual variance, and Eq 2.31 will somewhat underestimate the number of samples required to achieve a particular error goal.

2.2.2 FFPilot: a sampling algorithm designed to take advantage of the optimizing equation

We wanted to be able to apply the optimizing equation to real biochemical networks, but to do so we need some prior knowledge of the system under study. As shown in Eq 2.31, two global and $2\cdot(N+1)$ phase-specific parameters are required in order to apply the optimization equation and thereby calculate the optimal value of n_i. The two global parameters, the margin of error ζ and the confidence interval z score z_α, are independent of the model being studied and are set according to the desired error goal. The other parameters, the ratio $\frac{V[\mathtt{w}_0]}{{w_0}^2}$ (from phase 0), the successful crossing probabilities p_i (from phases $i > 0$), and the per-sample computational costs c_i, are model dependent and vary for each different combination of model, order parameter, and interface placement.

In general the exact values of the model-dependent parameters are unknown and estimates must be used instead. Rather than simplifying using assumptions such as constant cost[10,12], we produce rough but conservative estimates of the necessary parameters using a pilot simulation. By conservative, we mean that the estimates, when plugged into the optimization equation, will be likely to give values of n_i that are at least as large as the true values. This condition ensures that simulations run with n_i trajectories per phase will produce results that are at

least as accurate as the specified error goal.

We call our new enhanced sampling protocol FFPilot. The basic concept of FFPilot is to break an FFS simulation up into two stages. First a pilot stage is executed (see supplemental Fig S1), from which the parameters required for the optimization equation are estimated. Then, based on the results of the optimization equation, a production stage is planned and executed, from which the actual simulation output is calculated.

The FFPilot algorithm proceeds as follows:

1. Specify an error goal in terms of a target margin of error. Optionally, the confidence interval (which defaults to 95%) can be specified as well. As with standard FFS, the user must specify an order parameter and interface placements.

2. Begin FFPilot pilot stage:

   (a) Set the pilot stage sample count n_{pilot} to a single fixed value. Throughout this paper we used a value of $n_{\text{pilot}} = 10^4$.

   (b) Run a complete FFS simulation, following the algorithm described in Sec 2.1.1. Unlike standard FFS, the number of samples to collect in each phase is determined by a blind optimization method.

      i. Phase 0 proceeds the same as in standard FFS, using $n_0 = n_{\text{pilot}}$ as the sample count.

      ii. In phases $i > 0$, trajectories are run until n_{pilot} successful forward flux events are observed. It can be shown that, for a relatively modest number of successes, this method constrains error to within 2% when estimating the individual phase weights (see Appendix 2.A.2 for complete details).

3. When the pilot stage is finished, estimate the values required for the optimization equation from the results of the pilot simulation. Use confidence intervals to form conservative estimates that, when plugged into the optimization equation, are likely to yield values of n_i that are as large or larger than those required for the error goal.

4. Begin FFPilot production stage:

   (a) Determine n_i, the number of samples to collect in each phase, based on the error goal and Eq 2.31, the FFPilot optimizing equation (as parameterized in step 3).

   (b) Run another complete FFS simulation using using the values of n_i calculated in step 4a.

5. Collect results from the production stage simulation and use/analyze them in the same way as would be done for standard FFS simulation. The results from the pilot stage are ignored for the purposes of calculating the final simulation results as sampling differences in states at the interfaces would introduce additional error.

In terms of the error in the final simulation results, the optimization equation is inaccurate in the low sample count limit. Therefore, we enforce a minimum floor (10^3) on the count of samples taken in each phase of the production stage.

The effectiveness of the pilot stage blind optimization method can be related to an earlier finding of Borerro *et al.*[12]. They showed that a fixed quantity of computational effort is optimally spent during an FFS simulation when the interfaces and the trajectory counts per phase are arranged in such a way that each interface encounters an equal flux of trajectories crossing them. Although we do not constrain the computational effort spent during our blind optimization, our approach produces equal flux across each interface as well.

2.2.3 Rare event model

We began our testing of FFPilot with a toy model of a barrier crossing process, which we refer to as the rare event model (REM). REM models a particle in a discrete potential field in which there are two metastable states, $\mathcal{A}$ and $\mathcal{B}$, connected by a transition path (see top of Fig 2.1). Particles in $\mathcal{A}$ have a constant propensity to initiate a transition by entering the transition path. Because of the constant propensity the waiting times in between transition attempts are exponentially distributed.

The transition path itself is composed of a sequence of N steps; particles enter the path at step 1. At each successive step, a particle will instantaneously either proceed to the next step with probability p_i, or fall back into $\mathcal{A}$ with probability $1 - p_i$. If the particle successfully passes the final step it enters $\mathcal{B}$. In effect, the particle's fate once it enters the transition path can be thought of as the outcome of N weighted coin flips. If all N coins land heads up, the particle completes the transition $\mathcal{A} \to \mathcal{B}$. Otherwise, the particle falls back into $\mathcal{A}$.

We designed REM to map precisely onto FFPilot sampling in order to directly test the validity of the assumptions and simplifications that were made in the derivation of the FFPilot optimizing equation (Eq 2.28). Simulations of REM can be carried out using either DS or FFPilot. To simulate a single replicate of a particle starting in $\mathcal{A}$ using DS, first the time until the particle leaves $\mathcal{A}$ is randomly selected from an exponential random variable according to the propensity of entering the transition path. The particle's behavior at each step is then randomly chosen, either falling back to $\mathcal{A}$ or proceeding to the next step according to the appropriate p_i. If the particle falls back into $\mathcal{A}$ the process is repeated until it successfully passes to $\mathcal{B}$. The total time the particle took to transition to $\mathcal{B}$ is the $\mathcal{A} \to \mathcal{B}$ first passage time for that trajectory.

To simulate a single replicate of a particle starting in $\mathcal{A}$ using FFPilot, first an order parameter and the interface positions must be chosen (see section Sec 2.1.1).

We chose the step number i as the order parameter, and placed the interfaces between each step i. The phase 0 weight is calculated by first drawing many samples of the $\mathcal{A}$ leaving time according to the propensity, and then taking the mean of those samples. The remaining phase weights are determined by repeatedly starting a particle at step i, randomly selecting if it continues on to the next step according to p_i, and then calculating the average probability of success from the observations. The pilot stage of FFPilot is accomplished by first running the phase 0 weight calculation n_{pilot} times, then running each phase $i > 0$ weight calculation until n_{pilot} success events are observed. The outcome of the pilot stage is then fed into the FFPilot optimizing equation Eq 2.31, and the results are used to determine how many samples to take during the FFPilot production stage. For the purposes of parameterizing Eq 2.31, the phase 0 relative variance and the per-phase costs c_i are all set equal to 1 (see Table 2.1 for complete parameters). $MFPT$ is then estimated as the product of the production stage phase weights.

We first studied the distribution of the $MFPT$ estimators. Taken together, Eqs 2.15, 2.16, and 2.19 describe the normal distribution that repeated estimation of $MFPT$ is expected to produce. In our derivation we have assumed that we are working in the high sampling limit for values of w_i and n_i of interest. To test this assumption, we performed $1.6 \cdot 10^5$ independent simulations of REM using both DS and FFPilot using a 1% error goal. Fig 2.2 shows the distributions from our simulations. The black line in each figure is the normal distribution with mean and variance given by Eqs 2.16 and 2.19. The binned $MFPT$ estimates from both the DS and the FFPilot REM simulations are in excellent agreement with the predicted normal distribution.

Next, we tested how well the FFPilot approach was able to control sampling error in simulations of REM. If the method works as expected, 95% of simulations should have errors at or below the FFPilot error goal. We executed 1000 FFPilot

simulations of REM at 3 different error goals (10%, 3.2%, and 1%). We used the full FFPilot algorithm to determine how many trajectories to start at each interface.

The percent errors of the $MFPT$ calculated in each of these simulations are shown in Fig 2.3. The percent errors were calculated relative to the analytically determined $MFPT$. As can be seen, the 95th error percentiles (marked by the red lines) are located along $x = y$, indicating that overall error in the $MFPT$ estimates was constrained to the error goal. REM has no source of error aside from sampling error, and under these conditions FFPilot precisely controls the total simulation error.

2.2.4 Self regulating gene model

We next tested FFPilot with a relatively simple biochemical network, the self regulating gene model (SRG)[14]. SRG models expression of a single protein A. A is produced though autocatalysis, and decays via a first order process (see Fig 2.1).

In the deterministic formulation of SRG, the rate of change in the quantity of protein A is:

$$\frac{dA}{dt} = k_{low} + \left(k_{high} - k_{low}\right) \frac{A^h}{k_{50}^h + A^h} - A \tag{2.32}$$

For a given set of parameters, the fixed points of the state space of SRG can be found by setting Eq 2.32 equal to 0 and solving for A. For all of the parameters we used in our simulations there are three fixed points, two stable and one unstable. One of the stable fixed points corresponds to a state with a low count of A, and the other corresponds to a state with high count of A.

To formulate SRG as a stochastic model, we use the chemical master equation (CME) (see Table 2.2 for complete reaction list). The CME models the probability for the system to be in any state. Additionally, fluctuations due to population noise can cause the system to transition back and forth between the low and high states. The $MFPT$ between the states is related to the entropic barrier separating them.

We wanted to study how the height of the barrier between the low A and high A states affects the accuracy of FFPilot. Towards this end, we parameterized 3 different variants of SRG with different barrier heights, and thus different $MFPT$ values. We call these three variants $\text{SRG}_{\text{h=2.4}}$, $\text{SRG}_{\text{h=2.3}}$, and $\text{SRG}_{\text{h=2.2}}$, after the Hill coefficient used in the protein A production rate law. We tuned the other parameters in the model in order to approximately balance the occupancy of the low A and high A states in each of the variants (see Table 2.3 for parameter values).

For all FFPilot simulations of SRG we used the count of protein A as the order parameter. We determined the positions of the interfaces by first placing λ_0 a quarter of the distance (in terms of the order parameter) from the lower fixed point to the intermediate fixed point, then λ_N three quarters of the distance from the lower fixed point to the upper fixed point. We then placed 11 more interfaces spaced evenly between λ_0 and λ_N.

Unlike REM, trajectories in the low A state do not cross λ_0 and enter the transition pathway with a fixed propensity. Instead, the propensity changes dynamically with the system state, giving rise to a complex distribution of waiting times in between crossing events. In deriving the FFPilot optimizing equation (more specifically, when deriving Eq 2.19), we assumed that the first two central moments of the phase 0 waiting time distribution (w_0 and $V\left[\mathbb{w}_0\right]$) exist. In order to test this assumption we executed simulations in which we only performed phase 0, collecting 10^6 crossing events for λ_0.

Fig 2.4 shows the phase 0 inter-event time distributions. The tail of each distribution is fit well by a single exponential distribution, but the distribution near 0 is not. We estimated the value of the relative variance, $\frac{V[\mathbb{w}_0]}{{w_0}^2}$, used in the phase 0 terms of the FFPilot optimizing equation (Eq 2.31) to be 6.80, 6.43, and 5.96 for $\text{SRG}_{\text{h=2.4}}$, $\text{SRG}_{\text{h=2.3}}$, and $\text{SRG}_{\text{h=2.2}}$, respectively.

Next, we examined how well the FFPilot approach was able to control sampling

error with respect to $MFPT$. We executed 1000 FFPilot simulations of $SRG_{h=2.4}$, $SRG_{h=2.3}$, and $SRG_{h=2.2}$ using error goals 1%, 3.2%, and 10%. We estimated $MFPT$ from each simulation, then found the percent error relative to the value estimated from a DS simulation executed with an error goal of 0.62%.

The errors are shown in Fig 2.5. The 95th error percentiles are again located precisely along $x = y$. The accuracy of the estimates show that FFPilot is able to control error in both phase 0 (regardless of the exact distribution of the inter-event times) and the remaining phases for SRG.

We also looked at the contribution of phase 0 to the overall cost of the pilot stage. Applying Eq 2.22 to values from Table 2.3, we found that for all variants of SRG phase 0 required around $\sim 35\%$ of the total simulation time. This finding is in contrast to the longstanding assumption in the FFS literature that phase 0 does not significantly contribute to the cost of a simulation and should therefore be extensively sampled.

2.2.5 Genetic toggle switch model

The last model we investigated using FFPilot was a more complex gene regulatory network, one of a family of systems commonly referred to as a genetic toggle switch (GTS)[15]. Our GTS has seven species that interact with one another via fourteen reactions, all of which are first or second order (see Table 2.4). GTS consists of a single piece of operator DNA, O. When O is not bound to anything it can produce either of two proteins, A and B. A and B can both decay, they can both form homodimers, and those dimers can both bind back to O. Only one dimer can bind to O at any given time. When O is bound to a dimer of either protein, it can only produce more of that same protein (see Fig 2.1).

The combination of positive feedback (of monomer production on dimer/operator binding) and negative feedback (of dimer/operator binding on production of the

competing monomer) gives GTS bistable dynamics[16]. The system as a whole switches between a state with high levels of the various forms of A and low levels of B, and a state with low levels of A and high levels of B.

For GTS we defined three order parameters. One, which we called Δ, is the difference between the total count of protein B and the total count of protein A. Another, which we called Σ, is the sum of the total count of protein A and the total count of protein B. The last, Ω, takes a value from [-1,1] based solely on the state of the single operator. In terms of the underlying species counts, the order parameters can be written as:

$$\Delta = B + 2B_2 + 2OB_2 - (A + 2A_2 + 2OA_2)$$

$$\Sigma = A + 2A_2 + 2OA_2 + B + 2B_2 + 2OB_2$$

$$\Omega = -OA_2 + OB_2$$

where A and B are the monomer counts, A_2 and B_2 are the dimer counts, and OA_2 and OB_2 are the dimer-operator complex counts. Equivalently, Ω can be said to have one of three categorical values:

$$\Omega \rightarrow \{OA_2, O, OB_2\}$$

Although all of our GTS simulations are based on a stochastic master equation formulation of the system, it is helpful to consider the more straightforward deterministic formulation when trying to understand the system's overall behavior (see supplemental Table S1 for the deterministic rate equations). For a given set of parameters, the fixed points of the deterministic formulation can be found. For all of the parameter sets we used in our simulations there are three fixed points in terms of Δ, two stable fixed points and one unstable fixed point. One of the stable fixed points corresponds to the state with a high level of A and a low level of B, and the

other stable fixed point corresponds to the state with a low level of A and a high level of B. We refer to these two states as $\mathcal{A}$ and $\mathcal{B}$, respectively.

We wanted to be able to tune the rarity of the $\mathcal{A} \rightarrow \mathcal{B}$ event without disrupting the overall dynamics of GTS. To do so, we added a relative protein turnover parameter θ. The rate constants of all of the expression and decay reactions for both A and B are multiplied by θ. Since θ does not affect the birth/death ratio of each protein, the steady state levels of both A and B are constant with respect to θ. However, θ does have a large effect on the rate of $\mathcal{A} \rightarrow \mathcal{B}$ switching. We used three different variants of GTS in our simulations, GTS$_{\theta=.1}$, GTS$_{\theta=1}$, and GTS$_{\theta=10}$, see Table 2.5 for complete parameters.

For all FFPilot simulations we used Δ as the order parameter. We tiled the state space in terms of Δ by placing λ_0 at $\Delta = -27$, λ_N at $\Delta = 27$, and then placing 11 more interfaces evenly spaced in between.

As with SRG, we were interested in the distribution of phase 0 inter-event times of GTS in order to establish the validity of Eq 2.31 with respect to GTS. Fig 2.6 shows the results of our phase 0 inter-event time distribution simulations. The phase 0 distributions of the different GTS variants differ a great deal, but interestingly their $\frac{V[\mathbb{w}_0]}{{w_0}^2}$ values (the ratio of moments required for Eq 2.31) are very similar. $\frac{V[\mathbb{w}_0]}{{w_0}^2}$ was found to be 8.15, 8.15, and 8.41 for GTS$_{\theta=.1}$, GTS$_{\theta=1}$, and GTS$_{\theta=10}$, respectively.

We next ran a test to examine how well the full FFPilot protocol was able to control sampling error in estimations of $MFPT$ of the $\mathcal{A} \rightarrow \mathcal{B}$ switching process. We executed 1000 FFPilot simulations of GTS$_{\theta=.1}$, GTS$_{\theta=1}$, and GTS$_{\theta=10}$ using error goals 1%, 3.2%, and 10%. We estimated $MFPT$ for each simulation, then found the percent errors relative to the results from high accuracy DS simulations of equivalent models, which were run with a 0.62% error goal.

The percent errors of the $MFPT$ estimates are shown in Fig 2.7. As can be

seen in the figure, the 95th error percentiles (marked by the red lines) are located somewhat above $x = y$, indicating that the overall errors in the estimated $MFPT$ values are above the desired errors. The 95th percentile lines do decrease along with error goal, implying that FFPilot partially but not completely controls error in simulations of GTS. The anomalous dispersion decreases as the height of the barrier between $\mathcal{A}$ and $\mathcal{B}$ in probability space decreases. This implies that the extra error is caused by a system dependent property and is not directly related to undersampling.

We again found that phase 0 contributed significantly to the cost of GTS simulations. From parameters listed in Table 2.5 and Eq 2.22, we calculated the share of total simulation time consumed by phase 0, which was found to be 19%, 23%, and 33% for GTS$_{\theta=.1}$, GTS$_{\theta=1}$, and GTS$_{\theta=10}$, respectively.

2.2.6 Interface landscape error in genetic toggle switch

We sought to understand the causes of the extra error in the GTS simulations. We chose the condition with the largest anomalous errors, GTS$_{\theta=10}$ executed with an error goal of 10%, and examined the phase weight estimates produced by each of the 1000 replicate simulations we had run with that condition. These phase weights are shown in the top half of Fig 2.8. The phase weights estimated by an equivalent FFPilot simulation run with an error goal of 0.1% are shown as dashed lines, and serve as a point of reference (there is no exact method for extracting the phase weights from a DS simulation). The dispersion of phase weight estimates around the reference weight is much greater in certain phases, especially phases 4 and 5. By itself, this is not an indication that FFPilot is failing to correctly estimate sampling counts for these phases. By design, FFPilot allows for different levels of dispersion in different phases when it is determining the optimal simulation plan.

In order to determine how much of the phase weight dispersion represents FFPilot functioning as intended and how much of the dispersion is truly anomalous, we calculated an optimal set of error goal targets for each simulation phase. From the FFPilot optimizing equation (Eq 2.31) we derived analytic expressions for the per-phase error goals:

$$\zeta_{i=0} = \zeta \sqrt{\frac{\sqrt{\frac{V[\mathbb{w}_0]c_0}{{w_i}^2}}}{\sqrt{\frac{V[\mathbb{w}_0]c_0}{{w_i}^2}} + \sum_{j=1}^{N} \sqrt{\frac{(1-p_j)c_j}{p_j}}}},$$
$$\zeta_{i>0} = \zeta \sqrt{\frac{\sqrt{\frac{(1-p_i)c_i}{p_i}}}{\sqrt{\frac{V[\mathbb{w}_0]c_0}{{w_i}^2}} + \sum_{j=1}^{N} \sqrt{\frac{(1-p_j)c_j}{p_j}}}}. \tag{2.33}$$

Just as with the overall error goal, in any given phase $100 \cdot \alpha\%$ percent of simulations will have a level of sampling error in the phase weight estimate at or below the relevant per-phase error goal ζ_i. We parameterized Eq 2.33 using the phase weight and phase cost estimates from the very high accuracy (0.1% error goal) FFPilot simulation of $\text{GTS}_{\theta=10}$ mentioned above.

The error goal targets we calculated are shown as the dashed lines in the bottom half of Fig 2.8. The phase weight percent errors (calculated relative to the estimates from the 0.1% error goal simulation) are shown as dots, and the red lines mark the 95th percentiles of the errors. If the red line and the dashed line overlap for a particular phase, it means that the error in this phase is dominated by sampling error, which FFPilot is able to completely account for. If the red line is above the dashed line, then there is more error occurring in that phase than was predicted by FFPilot, and the magnitude of the separation of the two lines represents the magnitude of the anomalous (as opposed to the predicted) error. Interestingly, FFPilot estimates the majority of phase weights to within the desired error goal. The extra error in the $MFPT$ estimate appears to be due primarily to extra error in only three of the phase weight estimates, those from phases 3-5. Further, the

bulk of the extra error is concentrated in just two of those phase weight estimates, those from phases 4 and 5. Interfaces λ_4 and λ_5 also happen to be the interfaces immediately preceding the transition midpoint.

We hypothesized that there must be some particular feature of the state space landscape of GTS that the simulations are exploring during phases 4 and 5 that is responsible for the anomalous error. We further reasoned that the same features that are responsible for what Allen and coworkers call landscape variance[10] could be related to the increased error. Here we define a new source of error, which we call landscape error, that is due to two factors: (1) misrepresentation of some regions of the state space in the set of trajectory starting points at λ_i, and (2) significant differences in $P(\lambda_i|\lambda_{i-1})$ as a function of trajectory starting state. The total probability factor $P(\lambda_i|\lambda_{i-1})$ that is measured in each phase $i > 0$ can be thought of as a mixture of many independent probabilities, one for each state along λ_{i-1}, weighted by the (normalized) count of times the state is used as a starting point for a phase i trajectory. If either of factors (1) or (2) occurs alone, $P(\lambda_i|\lambda_{i-1})$ will still be correctly estimated. However, if the factors occur together they can lead to significant errors. In other words, if the landscapes assembled at λ_3 and λ_4 are heterogeneous across replicate simulations, and if differences in those landscapes can lead to differences in the effective value of $P(\lambda_i|\lambda_{i-1})$, then landscape error may be the cause of the anomalous simulation error we observe in our simulations of GTS. Of particular importance is the fact that the landscape error in phase i is due to errors in the landscape assembled from the endpoints of successful trajectories launched during phase $i-1$. Thus, no amount of extra sampling performed during phase i alone can completely abolish landscape error.

We wanted to test if the conditions for landscape error were present in FFPilot simulations of GTS$_{\theta=10}$. To this end, we chose two simulations, which we will call replicate 6 and replicate 8, that had a large divergence in their $MFPT$ estimates.

Looking at the phase weights estimates produced by these two simulations (Fig 2.9), the divergence in the $MFPT$ estimates can be seen to be mostly due to divergence in the phase weight 4 and 5 estimates (as expected). Exploring phase 5 in greater depth, we calculated the landscape occupancy along λ_4 in terms of the orthogonal order parameter Σ. The λ_4 occupancies for replicates 6 and 8 $P(\Sigma|\Omega, \lambda_4)$ (which are binned by Σ and operator state Ω), are shown in the lower left-hand subplots of Fig 2.10 as lines colored blue or gold, respectively. Replicate 6 has somewhat higher occupancy in the OA_2 and O states, and replicate 8 has higher occupancy in the OB_2 states. Thus, condition (1) for landscape error in phase 5, heterogeneous occupancies along λ_4, is indeed satisfied.

Next, we launched 10^6 independent trajectories from each starting state along λ_4 that had non-zero occupancy in either replicate 6 or 8. Just as in a normal FFPilot simulation, we stopped each trajectory when it either reached the next interface or fell back into the initial basin, and we took note of the stopping states. This gave us a highly accurate estimate of $P(\lambda_5|\lambda_4)$ as a function of trajectory starting state. These state-dependent probability values $P(\lambda_5|\Sigma, \Omega, \lambda_4)$, (which are also binned by Σ and operator state Ω), are shown as filled circles in Fig 2.10. $P(\lambda_5|\Sigma, \Omega, \lambda_4)$ varies a great deal across both Σ and Ω, meaning that condition (2) is also satisfied for our simulations of GTS$_{\theta=10}$, and that landscape error is indeed a possible explanation for the anomalous error.

In order to test if landscape error alone is a sufficient explanation for the observed anomalous error, we recalculated the phase 4 and 5 weights of replicates 6 and 8 from their landscapes alone, according to:

$$P(\lambda_5|\lambda_4) = \sum_{\Sigma} \sum_{\Omega} \left[P(\lambda_5|\Sigma, \Omega, \lambda_4) \times P(\Sigma, \Omega|\lambda_4) \right]. \tag{2.34}$$

If landscape error is indeed the sole cause of the anomalous error, we expected that the phase weights derived from Eq 2.34 would closely match those originally estimated by replicates 6 and 8, even though these original estimates vary greatly

between the replicates. In this view, the phase $i > 0$ weight estimate produced by each simulation converges (with increasing trajectory count) to a unique value of $P(\lambda_i|\lambda_{i-1})$, as determined by their heterogeneous samples of the landscape along λ_{i-1}. The recalculated phase weight values are plotted as empty circles in Fig 2.9, and they do indeed closely agree with the original estimates. Thus, we conclude that sampling error is indeed being handled correctly by FFPilot, and the anomalous error in our GTS simulation results is due to landscape error.

2.2.7 Eliminating landscape error in GTS via oversampling

Although a complete mathematical treatment of landscape error is beyond the scope of this paper, we wanted to investigate strategies for eliminating landscape error within the limited context of GTS. To this end we ran FFPilot simulations of $\text{GTS}_{\theta=10}$ under a variety of different oversampling schemes. As can be seen in Fig 2.8, for $\text{GTS}_{\theta=10}$ landscape error is mainly an issue in phases 3-5. Based on this, we initially we hypothesized that increasing sampling by 10X in phases 2-4 (*i.e.* increasing sampling in each of the phases preceding the problematic phases) would abolish landscape error.

The results from 1000 simulations of $\text{GTS}_{\theta=10}$ with 10X phase 2-4 sampling at an error goal of 10% are shown in the second column of Fig 2.11. Oversampling in these phases alone has only a minor effect on simulation error. Under these conditions the 95th percentile of error was 38%, whereas the error without oversampling is 47%. Based on these results, we tried out a more expansive oversampling strategy. In addition to oversampling by 10X in phases 2-4, we oversampled by 20X in phase 0 and 10X in phase 1 as well. The results from 1000 simulations run with 20X phase 0, 10X phase 1-4 oversampling are shown in the last column of Fig 2.11. Error is reduced dramatically under these conditions, to just under 10%. We also tried 10X phase 0 oversampling, but found that it was not quite sufficient to

eliminate landscape error (see supplemental Fig S5).

Thus, oversampling in phases 2-4 alone had almost no effect, but oversampling in phases 0-4 was enough to eliminate the landscape error. In order to understand this difference, we eliminated oversampling in each phase individually. The results from these simulations are shown in supplemental Fig S6. The effect of skipping oversampling in phase 0 is particularly dramatic, leading to an increase in simulation error of nearly 25%. So long as phase 0 is being oversampled, the increase in simulation error from skipping oversampling in any of phases 1-4 is less dramatic (2%-7%) but still significant. This implies that landscape errors are correlated. Effectively, defects in the sampled landscape distribution at any λ_i may carry over to λ_{i+1}.

2.2.8 Theoretical efficiency of DS, FFS, and FFPilot simulations

Enhanced sampling is commonly assumed to be more efficient than DS. By controlling for simulation error, a direct comparison can be made between DS and FFPilot simulations, and the speedup of one simulation method versus the other can be assessed.

It is straightforward to derive an expression for the cost of a DS simulation $\mathcal{C}_{\text{ds}}$ as a function of the error goal. Plugging Eq 2.12 into Eq 2.11 yields:

$$\mathcal{C}_{\text{ds}} = MFPT \frac{{z_\alpha}^2}{\zeta^2}. \tag{2.35}$$

The cost of an FFS simulation $\mathcal{C}_{\text{ffs}}$ is given by Eq 2.22: $\mathcal{C}_{\text{ffs}} = \sum_{i=0}^{N} n_i \; c_i$. We can plug the FFPilot optimizing equation (Eq 2.31) into Eq 2.22 in order to expand the n_i values. This yields an expression for $\mathcal{C}_{\text{ffs-opt}}$, the cost of an optimized FFS simulation given *a priori* knowledge of the parameters required for the optimizing equation:

$$\mathcal{C}_{\text{ffs-opt}} = \frac{{z_\alpha}^2}{\zeta^2} \left(\sqrt{\frac{c_0 \; V\left[\mathbb{w}_0\right]}{{w_0}^2}} + \sum_{i=1}^{N} \sqrt{\frac{c_i \; (1-p_i)}{p_i}} \right)^2. \tag{2.36}$$

Next, we examined the theoretical efficiency of the FFPilot approach. Due to the FFPilot optimizing equation, the production stage of an FFPilot simulation can be thought of the most computationally efficient FFS simulation possible with respect to a given error goal. However, if the pilot stage is too expensive it may be possible that in general FFPilot is inefficient relative to the traditional FFS algorithm. Thus we wanted to determine if FFPilot simulation is reasonably efficient, and, if so, under what conditions.

The total cost of an FFPilot simulation $\mathcal{C}_{\text{ffpilot}}$ can be found by adding a second term to the RHS of Eq 2.36 that specifically accounts for the extra runs performed during the pilot stage:

$$\mathcal{C}_{\text{ffpilot}} = \frac{{z_\alpha}^2}{\zeta^2}\left(\sqrt{\frac{c_0\, V\,[\mathbb{w}_0]}{{w_0}^2}} + \sum_{i=1}^{N}\sqrt{\frac{c_i\,(1-p_i)}{p_i}}\right)^2 + n_{\text{pilot}}\left(c_0 + \sum_{j=1}^{N}\frac{c_j}{p_j}\right). \quad (2.37)$$

Eq 2.37 gives $\mathcal{C}_{\text{ffpilot}}$ as a function of error goal. The production stage cost increases with error goal $\frac{\zeta^2}{{z_\alpha}^2}$ whereas the pilot stage cost remains fixed. For a low enough error goal, the pilot stage term in $\mathcal{C}_{\text{ffpilot}}$ will be negligible compared to the overall simulation cost. For simulations run with $n_{\text{pilot}} = 10^4$ the approximation $\mathcal{C}_{\text{ffpilot}} \approx \mathcal{C}_{\text{ffs-opt}}$ holds when the error goal was <5% (see Fig 2.12).

2.2.9 Speedup of FFPilot vs DS in simulations with equivalent error

We determined the speedup of FFPilot vs DS both as a function of simulation error and as a function of switching event rarity. We also examined the model dependence of the speedup. For SRG the speedup of FFPilot over DS is considerable and scales directly with switching event rarity (see top of Fig 2.12 and supplemental Fig S9). Given a 1% error goal, for $\text{SRG}_{\text{h=2.2}}$ the speedup of FFPilot over DS is 44X, whereas for $\text{SRG}_{\text{h=2.4}}$ the speedup increases to 323X.

For GTS, the FFPilot vs DS speedup results are somewhat more complicated to interpret, due to the extra landscape error in FFPilot. In order to make a fair

comparison between FFPilot and DS simulations of GTS, we oversampled our FFPilot simulations (see Sec 2.2.7) such that the observed error matched the error goal. The extra computational cost imposed by the oversampling must be taken into account when considering the speedup. The cost of an oversampled FFPilot simulation can be found by multiplying each sample count by its oversampling factor.

As with SRG, the simulation times of GTS simulations, both FFPilot and DS, were found to scale with switching event rarity roughly as a power law (see bottom of Fig 2.12 and Fig 2.13). If the effects of landscape error are ignored, then the FFPilot over DS speedups are as considerable for GTS as for SRG. Given a 1% error goal, the speedup for $\text{GTS}_{\theta=10}$ is 21X, while the speedup for $\text{GTS}_{\theta=.1}$ is 647X. However, if the oversampling required to correct for landscape error in GTS simulations is taken into account, the FFPilot speedup shrinks. With oversampling, given a 1% error goal the speedup for $\text{GTS}_{\theta=10}$ is 2X, while the speedup for $\text{GTS}_{\theta=.1}$ is 80X.

Supplementary material

See supplementary material for additional derivations and figures.

Acknowledgments

The authors thank the members of Roberts lab for discussions. This work was supported by the National Science Foundation under grant number PHY-1707961, and by the National Institutes of Health under grant T32 GM008403.

Appendices

2.A.1 The upper bound of the variance of a sum of Bernoulli distributions

The overall process of launching trajectories in order to determine the phase weight during each phase of FFS simulation can be thought of as a single draw from a sum of many independent Bernoulli distributions, also called a Poisson binomial distribution (PBD). This viewpoint emphasizes the statistical equivalence between the outcome of the jth trajectory in an FFS phase, which will either fall back into its starting basin or flux forward to the next interface, and the outcome of the jth Bernoulli random variable in a PBD, which will take on a value of either 0 or 1. In both cases "success" (fluxing forward, drawing a 1) occurs with some probability p_j inherent to the individual process, while "failure" (returning to basin, drawing a 0) occurs with probability $(1 - p_j)$. The expected value and the variance of a draw from a PBD of n terms (*i.e.* the sum over an independent draw from each of its constituent Bernoulli random variables) is:

$$\begin{aligned} \mu &= \sum_{j=1}^{n} p_j, \\ V &= \sum_{j=1}^{n} \left(1 - p_j\right) p_j. \end{aligned} \tag{2.38}$$

Equivalently, Eq 2.38 is also the mean and variance of the total count of successful trajectories n_i^s in FFS phase $i > 0$, given that $n = n_i$ trajectories were run.

The variance of a PBD is maximized when the probability parameter of each of its Bernoulli random variables are all the same p:

$$p_1 = p_2 = ... = p_n = p,$$

and so the upper bound on the variance of a sum of Bernoulli random variables is:

$$V \leq np(1 - p). \tag{2.39}$$

Proof. For a given value of μ, the method of Lagrange Multipliers can be used to maximize V with respect to the choice of particular values of the p_i terms. Appropriate constraint and target equations can be taken from Eq 2.38:

$$\begin{aligned} g\left[p_j\right] &= \sum_{j=1}^{n} p_j - \mu = 0 \\ f\left[p_j\right] &= V = \sum_{j=1}^{n} \left(1 - p_j\right) p_j. \end{aligned} \tag{2.40}$$

A Lagrangian can be formed from Eq 2.40:

$$\mathcal{L} = \sum_{j=1}^{n} \left(\left(1 - p_j\right) p_j\right) - \lambda \left(\mu - \sum_{k=1}^{n} p_k\right).$$

Next we find the gradient of the Lagrangian:

$$\partial_\lambda \mathcal{L} = \sum_{j=1}^{n} p_j - \mu$$

$$\partial_{p_j} \mathcal{L} = -2p_j + \lambda + 1.$$

Now we set each part of the gradient to zero and solve the resulting set of equations. We start by solving for p_i in terms of λ:

$$p_j = \frac{\lambda + 1}{2}.$$

Next we solve for λ alone:

$$\lambda = \frac{2\mu}{n} - 1.$$

Then finally we plug the solution for λ into the gradient of p_i and solve for p_i alone:

$$p_j = \frac{\mu}{n} = p. \tag{2.41}$$

Thus, the spot at which every p_i is equal to $\frac{\mu}{n}$ is a critical point for the variance of a Bernoulli mixture distribution. It can further be shown that the above critical point is a maximum for the constrained variance using the Bordered Hessian variation of

the classical second derivative test. The Bordered Hessian[17] of a Lagrangian can be defined as:

$$H \equiv \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial \lambda^2} & \frac{\partial \mathcal{L}}{\partial \lambda p_1} & \frac{\partial \mathcal{L}}{\partial \lambda p_2} & \cdots & \frac{\partial \mathcal{L}}{\partial \lambda p_N} \\ \frac{\partial \mathcal{L}}{\partial \lambda p_1} & \frac{\partial \mathcal{L}}{\partial p_1^2} & \frac{\partial \mathcal{L}}{\partial p_1 p_2} & \cdots & \frac{\partial \mathcal{L}}{\partial p_1 p_N} \\ \frac{\partial \mathcal{L}}{\partial \lambda p_2} & \frac{\partial \mathcal{L}}{\partial p_1 p_2} & \frac{\partial \mathcal{L}}{\partial p_2^2} & \cdots & \frac{\partial \mathcal{L}}{\partial p_2 p_N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{L}}{\partial \lambda p_N} & \frac{\partial \mathcal{L}}{\partial p_1 p_N} & \frac{\partial \mathcal{L}}{\partial p_2 p_N} & \cdots & \frac{\partial \mathcal{L}}{\partial p_N^2} \end{pmatrix},$$

which for our Lagrangian works out to:

$$H = \begin{pmatrix} 0 & 1 & 1 & ... & 1 \\ 1 & -2 & 0 & ... & 0 \\ 1 & 0 & -2 & ... & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 0 & ... & -2 \end{pmatrix}. \tag{2.42}$$

$\frac{\mu}{n}$ is a maximum if and only if H is negative definite. The negative definiteness of H can be demonstrated by showing that the signs of its leading principal minors demonstrate the appropriate alternating pattern[17]. For any finite value of n, this Hessian can be diagonalized using the Gaussian Elimination technique. This makes it easy to calculate the determinants of the various $x \times x$ upper-left submatrices and to show that the signs of said determinants do indeed follow the pattern of $(-1)^{x-1}$ for $x \geq 3$, satisfying the condition for negative definiteness. This conclusion can be generalized to arbitrary values of n using a proof by induction (the details of which are omitted for brevity). Thus, H is always a negative definite matrix, and so setting each $p_j = \frac{\mu}{n} = p$ does indeed maximize V for a given μ.

From Eq 2.39, the upper bound on the variance of the count of successful trajectories n_i^s in FFS phase $i > 0$ is:

$$V\left[n_i^s\right] \leq n_i p_i (1 - p_i).$$

From Eq 2.3, the phase weight estimator $\widehat{w}_{i>0}$ can be given in terms of the success count n_i^s:

$$\widehat{w}_{i>0} = \frac{n_i^s}{n_i}.$$

The variance of a quotient $V[x/y]$ is $V[x]/y^2$ given that y is a constant[11]. Thus, the upper bound on the variance of $\widehat{w}_{i>0}$ is:

$$V[\widehat{w}_{i>0}] = V\left[\frac{n_i^s}{n_i}\right] = \frac{V[n_i^s]}{n_i^2} \leq \frac{1}{n_i} p_i(1-p_i).$$

Finally, Eq 2.14 can be used to derive an upper bound on $V[\mathtt{w}_{i>0}]$ from the bound on $V[\widehat{w}_{i>0}]$:

$$V[\mathtt{w}_{i>0}] = n_i V[\widehat{w}_{i>0}] \leq p_i(1-p_i). \tag{2.43}$$

Parameterizing the FFPilot optimizing equation (Eq 2.31) with a value of $V[\mathtt{w}_{i>0}]$ that is at least as large as the true value helps to ensure that the calculated n_i is at least sufficient to achieve the given error goal. The upper bound on the variance given in Eq 2.43 is equivalent to the variance of a single Bernoulli random variable with parameter p_i. Thus, for the purposes of parameterizing the FFPilot optimizing equation we treat each $\mathtt{w}_{i>0}$ as a single Bernoulli random variable without any loss in accuracy.

2.A.2 Blind optimization method

Taken as a whole, the FFPilot approach to optimizing simulations can function reliably only if the results of the pilot stage are highly accurate (at least in terms of the individual parameter estimates) and computationally inexpensive (relative to the production stage). No prior knowledge of the system under study is used in the setup of the pilot stage. Thus, a "blind" optimization method, one that uses no information about the current phase or any other, must be used during this initial stage.

The blind optimization method that we use during the FFPilot pilot stage works by altering the conditions under which a simulation phase is terminated. During a standard FFS simulation, simulation phase $i > 0$ is terminated once a fixed number of trajectories n_i have launched from λ_{i-1} and have ended, regardless of where

(in state space) those trajectories have ended. During a pilot stage, we instead terminate phase $i > 0$ only once a fixed number $n_i^s = n_{\text{pilot}}$ of *successful* trajectories (*i.e.* the ones that reach λ_i) have been observed.

The advantage of using our blind optimization method is that it is able to produce estimates of the phase weight $w_{i>0} = p_i$ with constrained maximum error. The margin of error for a single phase $i > 0$ can be calculated from Eqs 2.21 and 2.29:

$$\zeta\left[\hat{p}_i\right] = z_\alpha \sqrt{\frac{1-p_i}{p_i n_i}}.$$

The total count of trajectories n_i required to produce n_{pilot} successful trajectories in phase $i > 0$ converges to $\frac{n_{\text{pilot}}}{p_i}$. Thus, with respect to the pilot stage the above equation can be rewritten as:

$$\zeta\left[\hat{p}_i\right] = z_\alpha \sqrt{\frac{1-p_i}{n_{\text{pilot}}}}. \tag{2.44}$$

The error increases as p_i becomes smaller, but it remains within a finite bound even as p_i goes to zero (see supplemental Fig S10). By default and throughout this paper we use the fixed values of $n_{\text{pilot}} = 10^4$ and $z_\alpha = z_{.95} \approx 1.96$ for all phases of the pilot stage. These values of n_{pilot} and z_α give a maximum $\hat{p}_i$ margin of error of 2%.

During phase 0, the distribution of samples taken from the underlying random variable $\mathbb{w}_0$ (*i.e.* the set of observed waiting times in between λ_0 forward crossing events) is model dependent. This means that no formulation equivalent to Eq 2.44 is possible for the phase weight $w_0 = \tau_{\mathcal{A}}$. However, the estimator $\hat{\tau}_{\mathcal{A}}$ can be in general assumed to be $\sqrt{n}$-consistent (as described in Secs 2.1.4 and 2.2.1.1). Plugging Eqs 2.9 and 2.14 into Eq 2.6 gives the asymptotic margin of error for phase 0 of the pilot stage:

$$\zeta\left[\hat{\tau}_{\mathcal{A}}\right] = \frac{z_\alpha}{\sqrt{n_{\text{pilot}}}} \frac{\sqrt{V\left[\mathbb{w}_0\right]}}{\tau_{\mathcal{A}}}. \tag{2.45}$$

It can be ensured that $\tau_{\mathcal{A}} \approx \sqrt{V\left[\mathbb{w}_0\right]}$ through appropriate placement of $\mathbb{w}_0$ (*i.e.* away from a basin of attraction). Given that $\tau_{\mathcal{A}}$ and $V\left[\mathbb{w}_0\right]$ are appropriately matched, the

margin of of error of phase 0 will be roughly proportional to $\frac{z_\alpha}{\sqrt{n_{\text{pilot}}}}$. For $n_{\text{pilot}} = 10^4$ and $z_\alpha \approx 1.96$, this also works out to a $\hat{\tau}_{\mathcal{A}}$ margin of error of about 2%. Thus, the blind optimization approach used in the FFPilot pilot stage controls error in $\hat{\tau}_{\mathcal{A}}$, though not in such a conveniently bounded fashion as $\hat{p}_i$.

Tables

Table 2.1: Parameterization and other simulation data from our simplified Rare Event Model (REM). The phase weights w_i were copied from those of $\text{GTS}_{\theta=1}$ (see Table 2.5). All sample costs c were set to 1.

phase	$^{\star}w_i$	$^{\dagger}\, c_i^f$	$^{\ddagger}\, c_i^s$	$^{\S}\, c_i$	$^{\P}n_i \times 10^{-3}$
0	45.3	—	—	1	1834
1	.092	1	1	1	2020
2	.274	1	1	1	1047
3	.136	1	1	1	1621
4	.150	1	1	1	1527
5	.242	1	1	1	1138
6	.636	1	1	1	486
7	.711	1	1	1	410
8	.858	1	1	1	261
9	.913	1	1	1	199
10	.973	1	1	1	107
11	.989	1	1	1	67
12	.998	1	1	1	30
MFPT	$1.074 \cdot 10^6$				

*The phase weight of phase i.

†The expected cost of a failed phase i trajectory.

‡The expected cost of a successful phase i trajectory.

§The expected cost of a phase i sample.

¶The phase i sample count (in thousands) for a 1% error goal with 95% confidence.

Table 2.2: Reaction scheme for our Self Regulating Gene (SRG) models.

Reactions	Rates	Description
$\emptyset \longrightarrow \mathsf{A}$	$k_{low} + \left(k_{high} - k_{low}\right) \frac{\mathsf{A}^h}{k_{50}^h + \mathsf{A}^h}$	expression
$\mathsf{A} \longrightarrow \emptyset$	$\beta \mathsf{A}$	decay

Table 2.3: Parameterizations and other simulation data from our SRG models. Per-phase weights, costs, and sample counts were estimated from the average results of 1000 FFPilot simulations (1% error goal). $MFPT$ values were estimated from the results of 10^5 DS simulations, plus or minus the 95% confidence interval bounds.

parameter	$SRG_{h=2.4}$						$SRG_{h=2.3}$						$SRG_{h=2.2}$					
h	2.4						2.3						2.2					
k_{low}	10						10						10					
k_{high}	200						200						200					
k_{50}	91						92.5						94					
β	1						1						1					
phase	$^{*}\lambda_i$	$^{\dagger}w_i$	$^{\ddagger}c_i^f$	$^{\S}c_i^s$	$^{\P}c_i$	$^{\|}n_i \times 10^{-3}$	λ_i	w_i	c_i^f	c_i^s	c_i	$n_i \times 10^{-3}$	λ_i	w_i	c_i^f	c_i^s	c_i	$n_i \times 10^{-3}$
0	23.0	9.66	—	—	9.66	641	23.0	6.89	—	—	6.89	703	23.0	4.78	—	—	4.78	762
1	33.6	.007	.130	.913	.135	26395	34.1	.010	.137	.936	.150	18975	34.6	.018	.138	.846	.151	12067
2	44.2	.071	1.56	1.21	1.53	2383	45.2	.112	1.65	1.20	1.60	1620	46.2	.140	1.64	1.30	1.60	1253
3	54.8	.277	2.89	1.14	2.41	850	56.2	.296	2.99	1.24	2.47	714	57.8	.354	3.05	1.23	2.41	556
4	65.3	.565	4.22	1.30	2.57	447	67.3	.580	4.35	1.28	2.57	387	69.3	.572	4.45	1.39	2.70	337
5	75.9	.855	5.45	1.12	1.75	254	78.4	.825	5.63	1.23	2.00	237	80.9	.814	5.78	1.23	2.08	212
6	86.5	.962	6.48	1.03	1.24	147	89.5	.946	6.75	1.08	1.39	147	92.5	.924	7.01	1.24	1.67	141
7	97.1	.993	7.35	.844	.890	73	101	.986	7.71	.926	1.02	85	104	.977	8.12	1.09	1.25	88
8	108	.999	8.11	.700	.709	34	112	.997	8.56	.836	.862	46	116	.993	9.07	.905	.959	53
9	118	1.00	8.83	.767	.769	16	123	.999	9.42	.825	.833	25	127	.997	10.0	.981	1.00	32
10	129	1.00	—	.779	.779	8	134	1.00	10.4	.894	.897	14	139	.999	11.0	.955	.965	20
11	139	1.00	—	1.05	1.05	5	145	1.00	—	1.07	1.07	9	150	.999	12.3	1.24	1.25	14
12	150	1.00	—	1.33	1.33	3	156	1.00	—	1.44	1.44	6	162	1.00	13.8	1.48	1.49	10
MFPT	$(1.521 \pm .0094) \cdot 10^5$						$(4.661 \pm .029) \cdot 10^4$						$(1.271 \pm .0079) \cdot 10^4$					

*The position of interface i in terms of order parameter A.
†The phase weight of phase i.
‡The expected cost of a failed phase i trajectory.
§The expected cost of a successful phase i trajectory.
¶The expected cost of a phase i sample.
‖The phase i sample count (in thousands) for a 1% error goal with 95% confidence.

Table 2.4: Reaction scheme for our Genetic Toggle Switch (GTS) models.

Reactions		Rate Constants		Description
Protein A	*Protein B*	*forward*	*reverse*	
$\mathsf{O} \longrightarrow \mathsf{O} + \mathsf{A}$	$\mathsf{O} \longrightarrow \mathsf{O} + \mathsf{B}$	θ		expression
$\mathsf{A} \longrightarrow \emptyset$	$\mathsf{B} \longrightarrow \emptyset$	$\theta/4$		decay
$2\,\mathsf{A} \rightleftharpoons \mathsf{A}_2$	$2\,\mathsf{B} \rightleftharpoons \mathsf{B}_2$	5	5	dimerization
$\mathsf{O} + \mathsf{A}_2 \rightleftharpoons \mathsf{OA}_2$	$\mathsf{O} + \mathsf{B}_2 \rightleftharpoons \mathsf{OB}_2$	5	1	operator binding
$\mathsf{OA}_2 \longrightarrow \mathsf{OA}_2 + \mathsf{A}$	$\mathsf{OB}_2 \longrightarrow \mathsf{OB}_2 + \mathsf{B}$	θ		bound expression

Table 2.5: Parameterizations and other simulation data from our GTS models. Per-phase weights, costs, and sample counts were estimated from the average results of 1000 FFPilot simulations (1% error goal). $MFPT$ values were estimated from the results of 10^5 DS simulations, plus or minus the 95% confidence interval bounds.

parameter		GTS$_{\theta=.1}$					GTS$_{\theta=1}$					GTS$_{\theta=10}$				
θ		.1					1					10				
phase	*λ_i	†w_i	‡ c_i^f	§ c_i^s	¶ c_i	‖$n_i \times 10^{-3}$	w_i	c_i^f	c_i^s	c_i	$n_i \times 10^{-3}$	w_i	c_i^f	c_i^s	c_i	$n_i \times 10^{-3}$
0	-27.0	453	—	—	453	1664	45.3	—	—	45.3	1337	4.52	—	—	4.52	985
1	-22.5	.092	19.7	63.5	23.7	8010	.092	1.97	6.35	2.37	6436	.090	.197	.643	.238	4698
2	-18.0	.271	148	113	138	1728	.274	14.8	11.2	13.8	1382	.281	1.51	1.12	1.40	977
3	-13.5	.128	307	176	290	1897	.136	30.7	17.1	28.8	1482	.170	3.13	1.46	2.85	946
4	-9.0	.111	442	120	406	1738	.150	44.0	10.5	38.9	1201	.339	4.46	.597	3.15	568
5	-4.5	.082	536	131	503	1849	.242	52.6	10.1	42.3	859	.676	4.98	.529	1.97	356
6	0.0	.437	659	118	423	684	.636	61.4	8.93	28.0	451	.872	5.14	.463	1.06	268
7	4.5	.632	830	167	411	467	.711	73.6	12.7	30.3	365	.873	5.83	.657	1.32	240
8	9.0	.861	994	140	259	310	.858	85.1	11.1	21.6	276	.918	6.43	.599	1.08	208
9	13.5	.940	1153	165	225	209	.913	97.3	14.4	21.6	210	.925	7.08	.861	1.33	178
10	18.0	.988	1283	122	135	116	.973	108	11.5	14.1	140	.960	7.58	.809	1.08	141
11	22.5	.997	1397	156	159	53	.989	117	15.4	16.5	81	.972	8.14	1.27	1.46	102
12	27.0	1.00	1493	150	150	19	.998	126	14.9	15.1	37	.989	8.61	1.36	1.44	63
	MFPT	$(7.015 \pm .043) \cdot 10^7$					$(1.074 \pm .0067) \cdot 10^6$					$(7.701 \pm .048) \cdot 10^3$				

*The position of interface i in terms of order parameter Δ.

†The phase weight of phase i.

‡The expected cost of a failed phase i trajectory.

§The expected cost of a successful phase i trajectory.

¶The expected cost of a phase i sample.

‖The phase i sample count (in thousands) predicted by FFPilot to achieve a 1% error goal, 95% confidence.

Figures

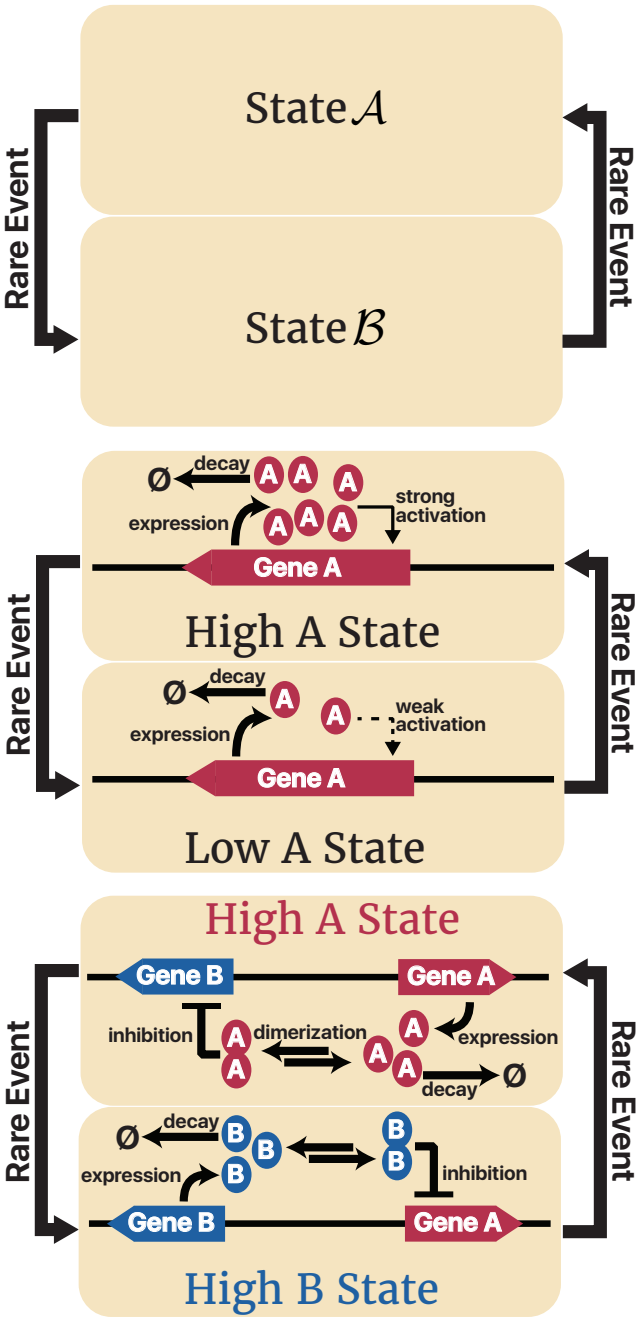


Figure 2.1: Schematics of the model systems investigated. (top) The Rare Event Model (REM, see Sec 2.2.3 and Table 2.1 for complete details). (middle) Self Regulating Gene (SRG, see Sec 2.2.4 and Tables 2.2 and 2.3). (bottom) Genetic Toggle Switch (GTS, see Sec 2.2.5 and supplemental Table S1.

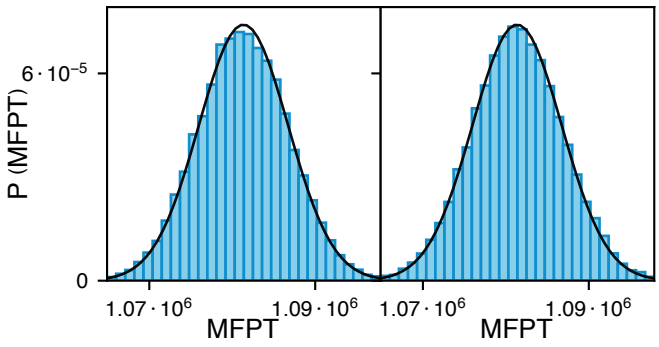


Figure 2.2: $MFPT$ estimates for REM calculated using (left) DS and (right) FFPilot. Shown are (blue bars) binned $MFPT$ estimates from $1.6 \cdot 10^5$ simulations with a 1% error goal and (black lines) $MFPT$ estimate distributions from Eq 2.7 and Eq 2.15, respectively.

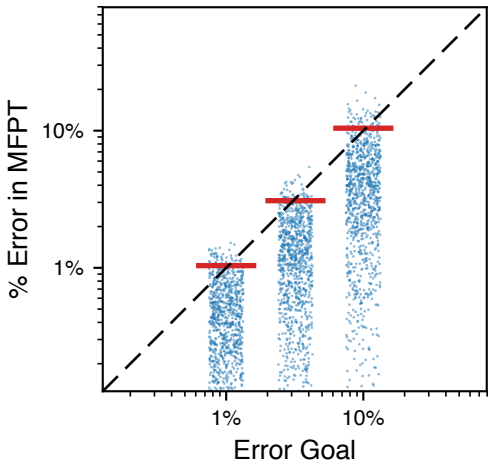


Figure 2.3: Actual error vs error goal of FFPilot simulations of the REM. Each strip shows the $MFPT$ estimation error from (blue dots) 1000 independent FFPilot simulations and (red lines) the 95th percentiles of the errors. Jitter was added to the x-position of the dots for visualization.

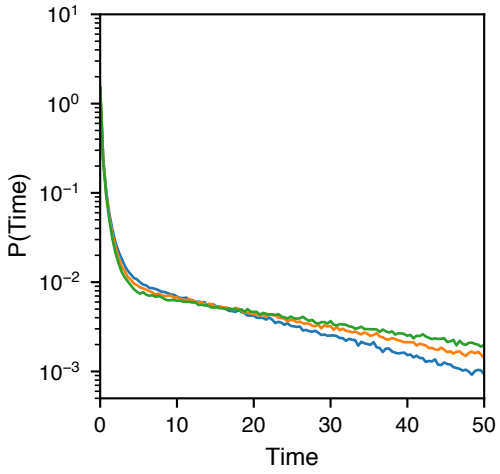


Figure 2.4: Waiting times between forward crossing events during phase 0 of forward flux simulations of the SRG models. Each line shows a distribution calculated from 10^6 samples from a single phase 0 trajectory of (blue) $SRG_{h=2.4}$, (orange) $SRG_{h=2.3}$, and (green) $SRG_{h=2.2}$.

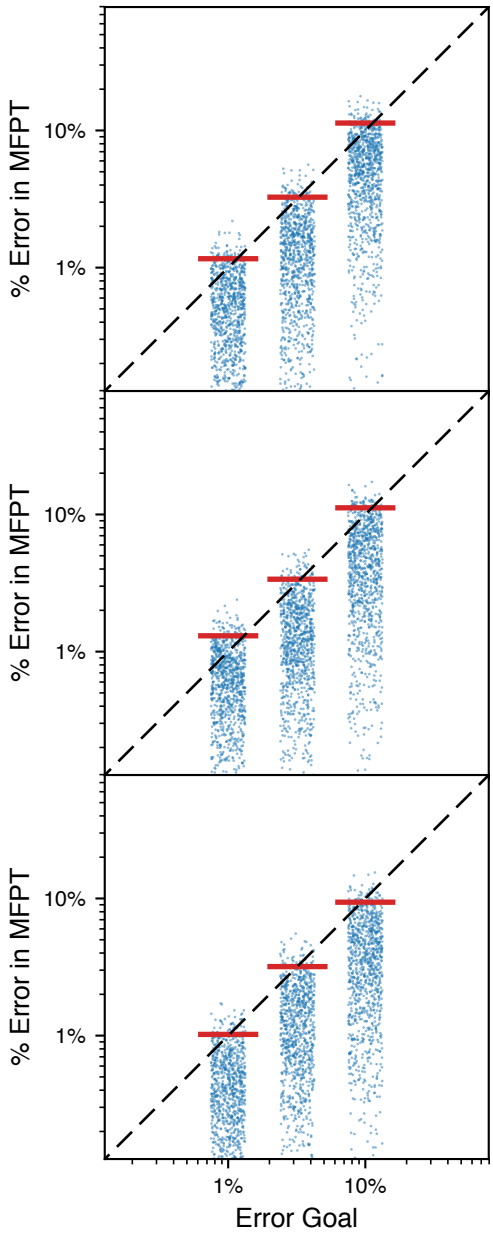


Figure 2.5: Actual error vs error goal of FFPilot simulations for (top) $SRG_{h=2.4}$, (middle) $SRG_{h=2.3}$, and (bottom) $SRG_{h=2.2}$. Each strip shows (blue dots) 1000 independent FFPilot simulations and (red lines) the 95th percentiles of the errors.

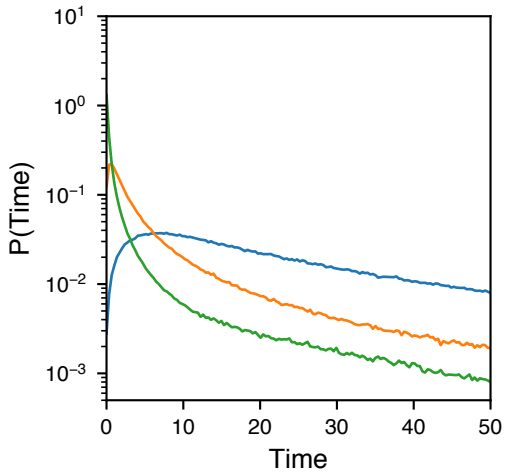


Figure 2.6: Waiting times between forward crossing events during phase 0 of FFS simulations of the GTS models. Each line shows a distribution calculated from 10^6 samples from a single phase 0 trajectory of (blue) $\text{GTS}_{\theta=.1}$, (orange) $\text{GTS}_{\theta=1}$, and (green) $\text{GTS}_{\theta=10}$.

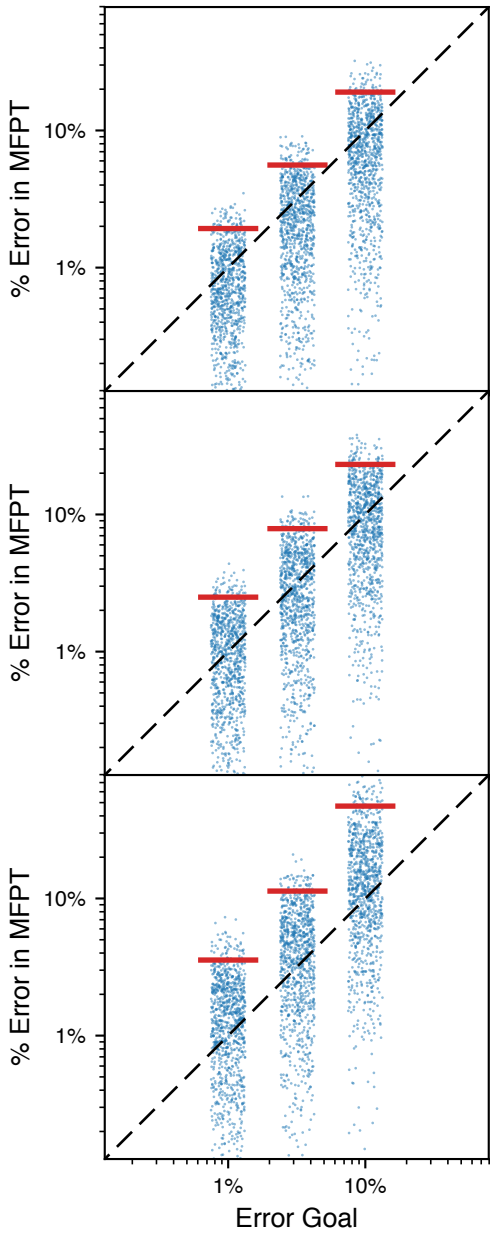


Figure 2.7: Actual error vs error goal of FFPilot simulations for (top) $\text{GTS}_{\theta=.1}$, (middle) $\text{GTS}_{\theta=1}$, and (bottom) $\text{GTS}_{\theta=10}$. Each strip shows (blue dots) 1000 independent FFPilot simulations and (red lines) the 95th percentiles of the errors.

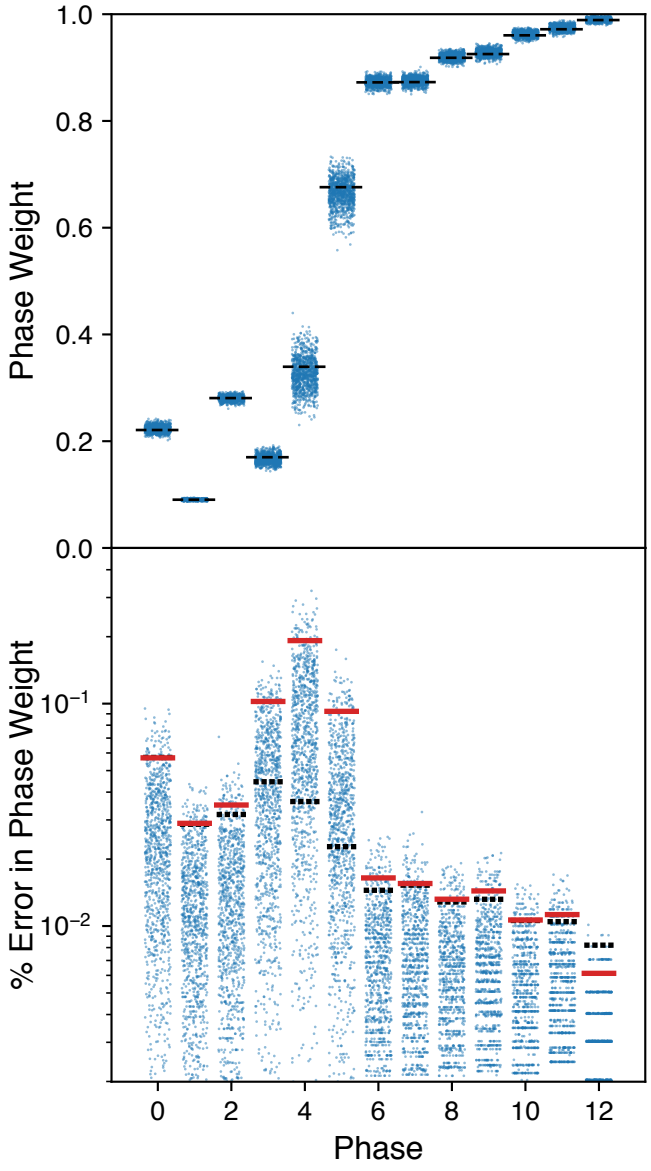


Figure 2.8: $\text{GTS}_{\theta=10}$ phase weight estimates and errors. (top) The phase weights as estimated by (blue dots) 1000 independent FFPilot simulations run to a 10% error goal. The dashed black lines show the phase weights from an FFPilot simulation with a 0.1% error goal. (bottom) The blue dots show the percent error of each phase weight estimate given in the top subplot, relative to the 0.1% error goal simulation. Also shown are (red lines) the observed 95th error percentiles, and (dashed black lines) the expected 95th error percentiles.

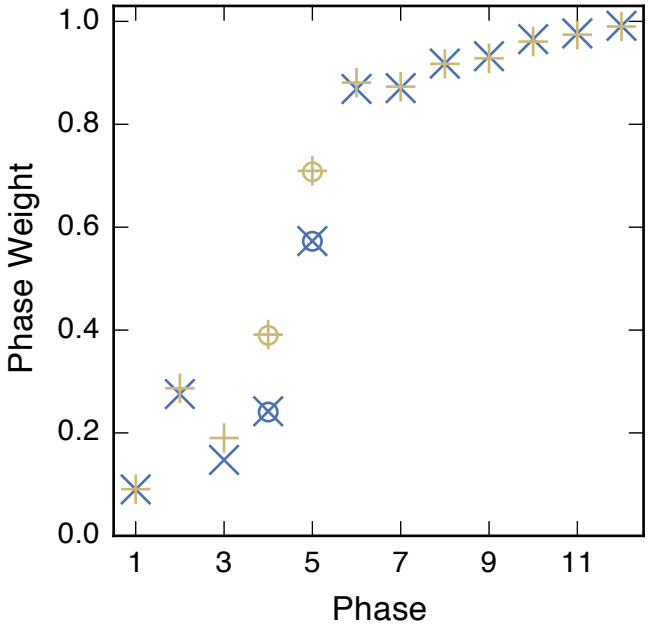


Figure 2.9: The phase weights of two replicates: (blue x) Replicate 6 and (gold +) Replicate 8, from a $GTS_{\theta=10}$ FFPilot simulation run to an error goal of 10%. Circles show weights for phases 4 and 5 calculated via an alternative method using Eq 2.34.

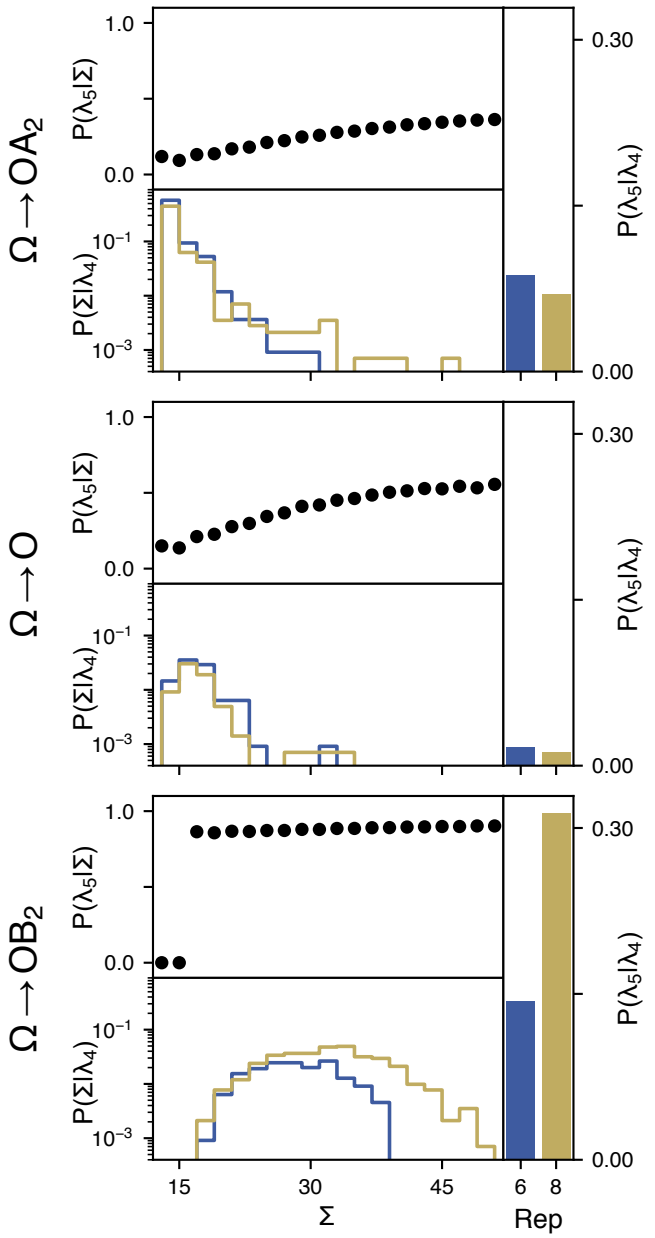


Figure 2.10: Breakout of the calculation of phase weight 5 for replicates 6 and 8. Each group of three plots show data from a slice of GTS state space with a different Ω, the state of the operator DNA. Each subplot within a group shows different probability data. (dots) The probability of a trajectory launched from a point on λ_4 fluxing forward to λ_5 vs the orthogonal order parameter Σ. (lines) The occupancy along Σ at interface λ_4 for (blue) replicate 6 and (gold) replicate 8. (bars) The cumulative probability of fluxing forward across λ_5 when starting at the specified Ω for (blue) replicate 6 and (gold) replicate 8.

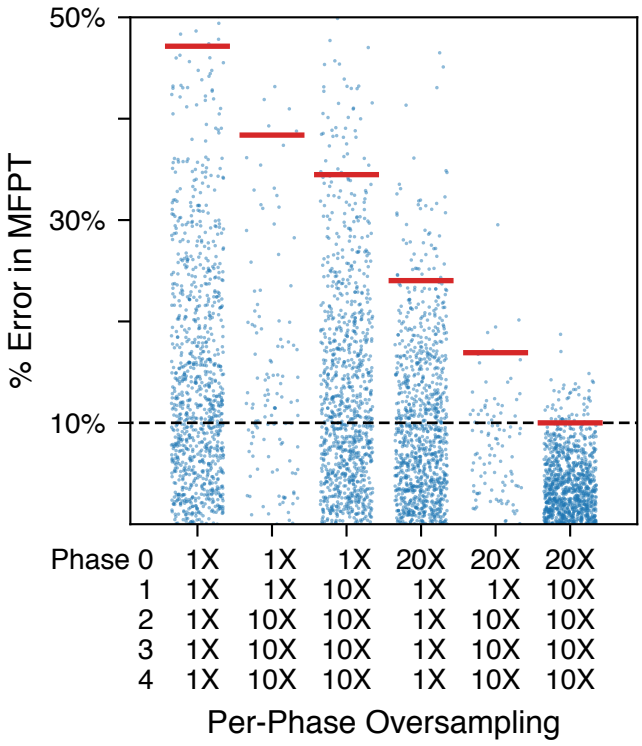


Figure 2.11: Errors from simulations executed with various oversampling schemes. The extent of oversampling in each phase for the separate schemes is indicated beneath each column of results. All simulations were of $\text{GTS}_{\theta=10}$ run to an error goal of 10%. Red lines show the 95th error percentiles.

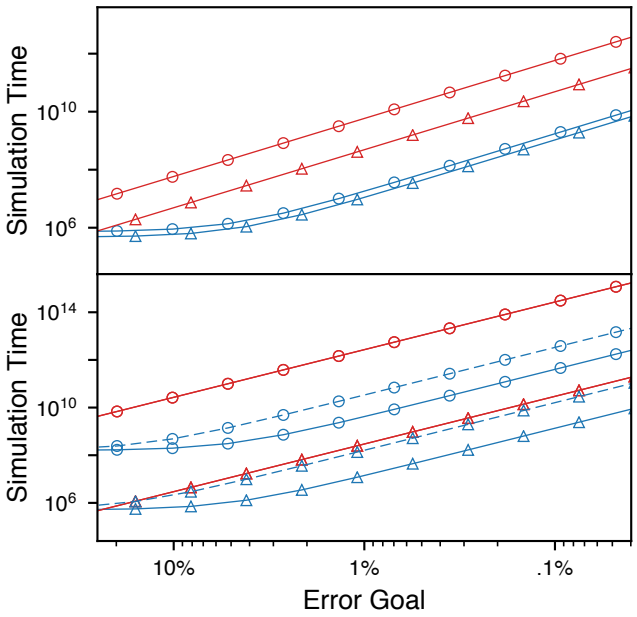


Figure 2.12: Simulation time vs error goal for several SRG and GTS models. Simulation time was calculated using Eq 2.35 for (red lines) DS, and Eq 2.37 for (blue lines) FFPilot and (dashed line) FFPilot with oversampling to correct the landscape error. Parameters were taken from Tables 2.3 and 2.5. (top) Data from (circles) $\text{SRG}_{\text{h=2.4}}$ and (triangles) $\text{SRG}_{\text{h=2.2}}$. (bottom) Data from (circles) $\text{GTS}_{\theta=.1}$ and (triangles) $\text{GTS}_{\theta=10}$.

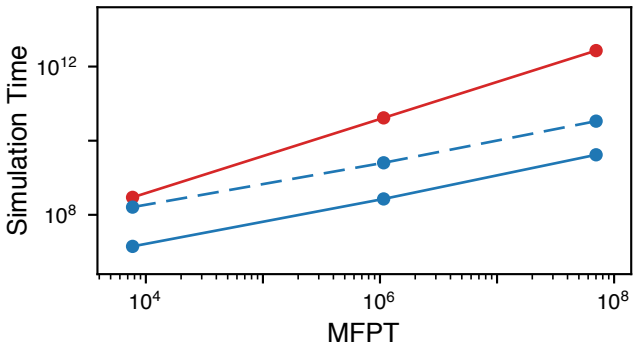


Figure 2.13: Simulation time vs $MFPT$ for several GTS models. Simulation times are shown for (red dots) DS, (blue dots) FFPilot, and (blue dots with dashed line) FFPilot with oversampling to correct for landscape error. The simulation times shown are averaged across 1000 simulations. Connecting lines added for illustration.

Supplemental

2.S.1 Alternative derivation of variance of the $MFPT$ estimator

Instead of using the delta method as in Sec 2.2.1.1 of the main paper, the variance of the FFS $MFPT$ estimator can also be derived from the fundamental properties of variance (though no explicit information about the underlying distribution of $\widehat{MFPT}$ is gained this way). From the general properties of variance it is known that for independent random variables $X_0, X_1, ..., X_i$:

$$V\left[\prod_i X_i\right] = \prod_i \left(V\left[X_i\right] + E\left[X_i\right]^2\right) - \prod_i E\left[X_i\right]^2. \tag{2.46}$$

The righthand side of the above Eq 2.46 can be rewritten in terms of a generalized variance G[18]:

$$V\left[\prod_i X_i\right] = \prod_i E\left[X_i\right]^2 \left(\sum_i G\left[X_i\right] + \sum_{i_0<i_1} G\left[X_{i_0}\right] G\left[X_{i_1}\right] + \sum_{i_0<i_1<i_2} G\left[X_{i_0}\right] G\left[X_{i_1}\right] G\left[X_{i_2}\right] + \right. \tag{2.47}$$

where $G\left[X\right] = \frac{V[X]}{E[X]^2}$. If the expected values are much larger than the variances, the higher order terms of $V\left[\prod_i X_i\right]$ can be ignored without a large loss of accuracy. The condition:

$$E\left[X_i\right] >> V\left[X_i\right] \quad \forall X_i, \tag{2.48}$$

implies that:

$$G\left[X_{i_0}\right] >> G\left[X_{i_0}\right] G\left[X_{i_1}\right] \qquad \forall X_{i_0}, X_{i_1},$$

$$G\left[X_{i_0}\right] >> G\left[X_{i_0}\right] G\left[X_{i_1}\right] G\left[X_{i_2}\right] \qquad \forall X_{i_0}, X_{i_1}, X_{i_2},$$

$$\vdots$$

This regime yields an easy to work with approximation of Eq 2.47 as a series of of independent terms that only depend on a single X_i:

$$V\left[\prod_i X_i\right] \approx \prod_i E\left[X_i\right]^2 \sum_i G\left[X_i\right]. \tag{2.49}$$

Each FFS simulation phase i can be conceptualized as taking a series of samples from a random variable $\mathbb{w}_i$ (see Sec 2.1.1 in the main paper). The phase weight estimator $\widehat{w}_i$ is then the mean of the $\mathbb{w}_i$ samples. The moments of each $\widehat{w}_i$ can be derived using the standard expected value and variance identities[19], giving:

$$\begin{aligned} E\left[\widehat{w}_i\right] &= w_i, \\ V\left[\widehat{w}_i\right] &= \frac{V\left[\mathbb{w}_i\right]}{n_i}. \end{aligned} \tag{2.50}$$

Since $V[w_i]$ is a fixed value, the value of each $V[\widehat{w}_i]$ term trends monotonically downward as n_i increases. If n_i is assumed to be set to a large value, then it is reasonable to assume that $E[\widehat{w}_i] >> V[\widehat{w}_i]$ as well. In this regime the condition in Eq 2.48 is satisfied, and so we can apply the simplified formula for the product variance to the phase weights. Plugging the moments in Eq 2.50 into Eq 2.49 yields:

$$V\left[\widehat{MFPT}\right] = V\left[\prod_i \widehat{w}_i\right] \approx \prod_i E[\widehat{w}_i]^2 \sum_i \frac{G[\widehat{w}_i]}{n_i} = \prod_j {w_j}^2 \sum_i \frac{V[\mathbb{w}_i]}{{w_i}^2 n_i}.$$

2.S.2 Speedup equation misc

We present here the complete FFPilot equation for the speedup of FFPilot over DS simulation, $\frac{\mathcal{C}_{\text{ds}}}{\mathcal{C}_{\text{ffpilot}}}$, without the high simulation accuracy/low error goal approximation used in the main paper:

$$\frac{\mathcal{C}_{\text{ds}}}{\mathcal{C}_{\text{ffpilot}}} = MFPT / \left(\left(\sum_{i=1}^{N} \sqrt{\frac{c_i\,(1-p_i)}{p_i}} + \sqrt{\frac{c_0\,V[\mathbb{w}_0]}{w_0^2}} \right)^2 + n_{pilot} \frac{\zeta^2}{{z_\alpha}^2} \left(c_0 + \sum_{j=1}^{N} \frac{c_j}{p_j} \right) \right).$$

2.S.3 Supplemental figures

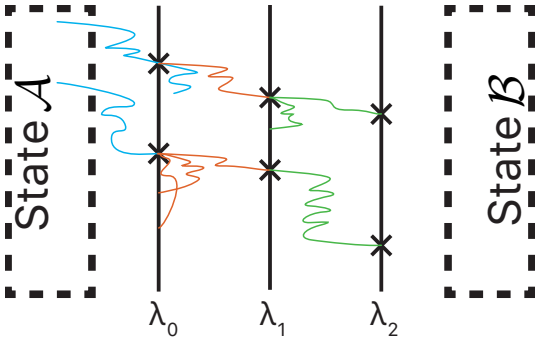


Figure S1: Schematic example of a pilot stage from a FFPilot simulation, with total phase count $N = 3$ and $n_{\text{pilot}} = 2$. Trajectories from phases (blue) 0, (red) 1, and (green) 2 shown in different colors.

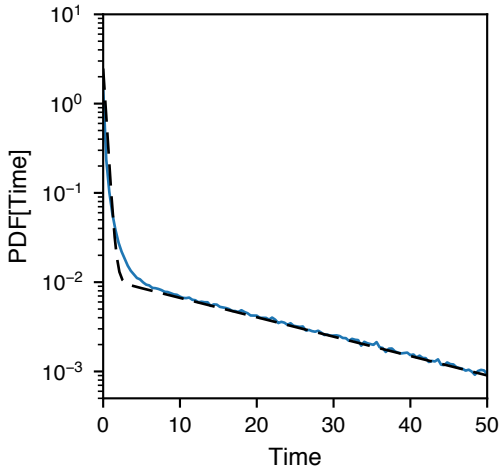


Figure S2: Distribution of waiting times in between phase 0 forward flux events for $\mathrm{SRG}_{\mathrm{h=2.2}}$, 10^6 samples. Dashed line is a fit of a mixture of two exponential distributions.

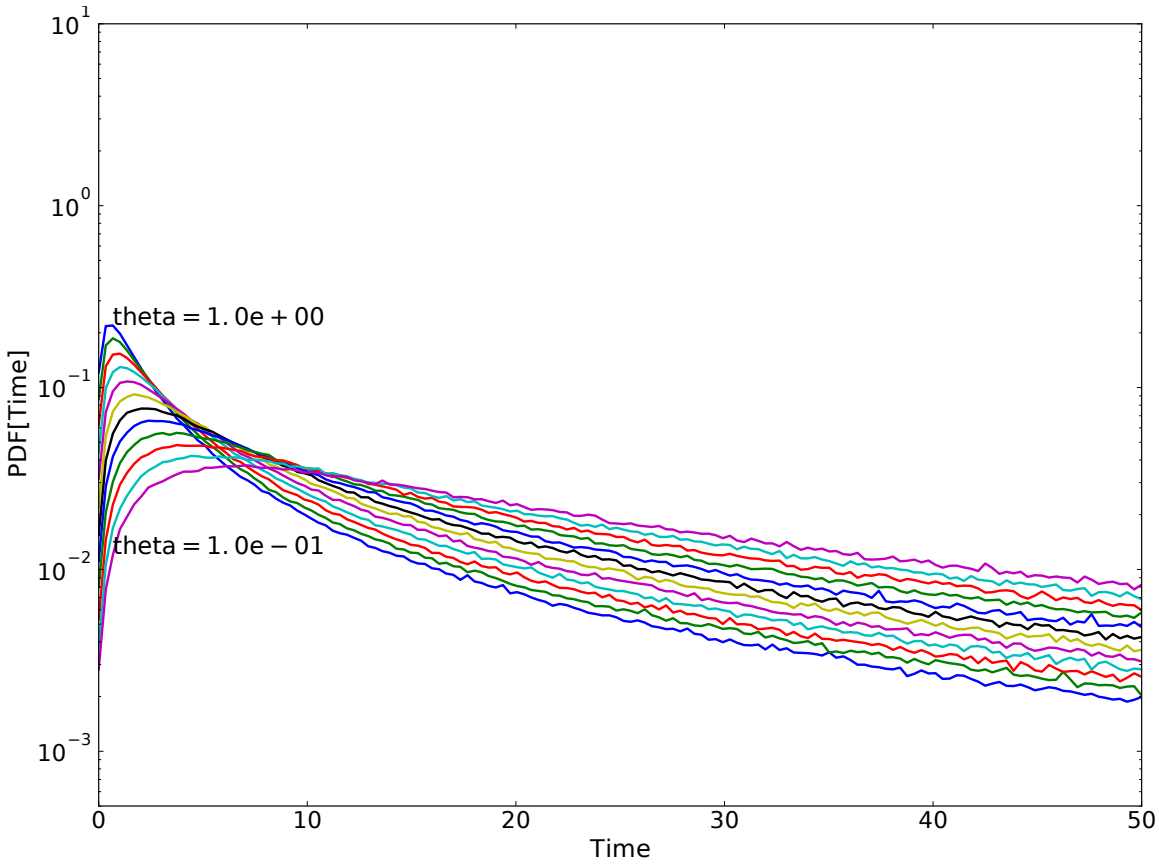


Figure S3: Genetic Toggle Switch phase 0 inter-event time distribution for values of θ in between $.1$ and 10 (inclusive). 10^6 samples in each distribution.

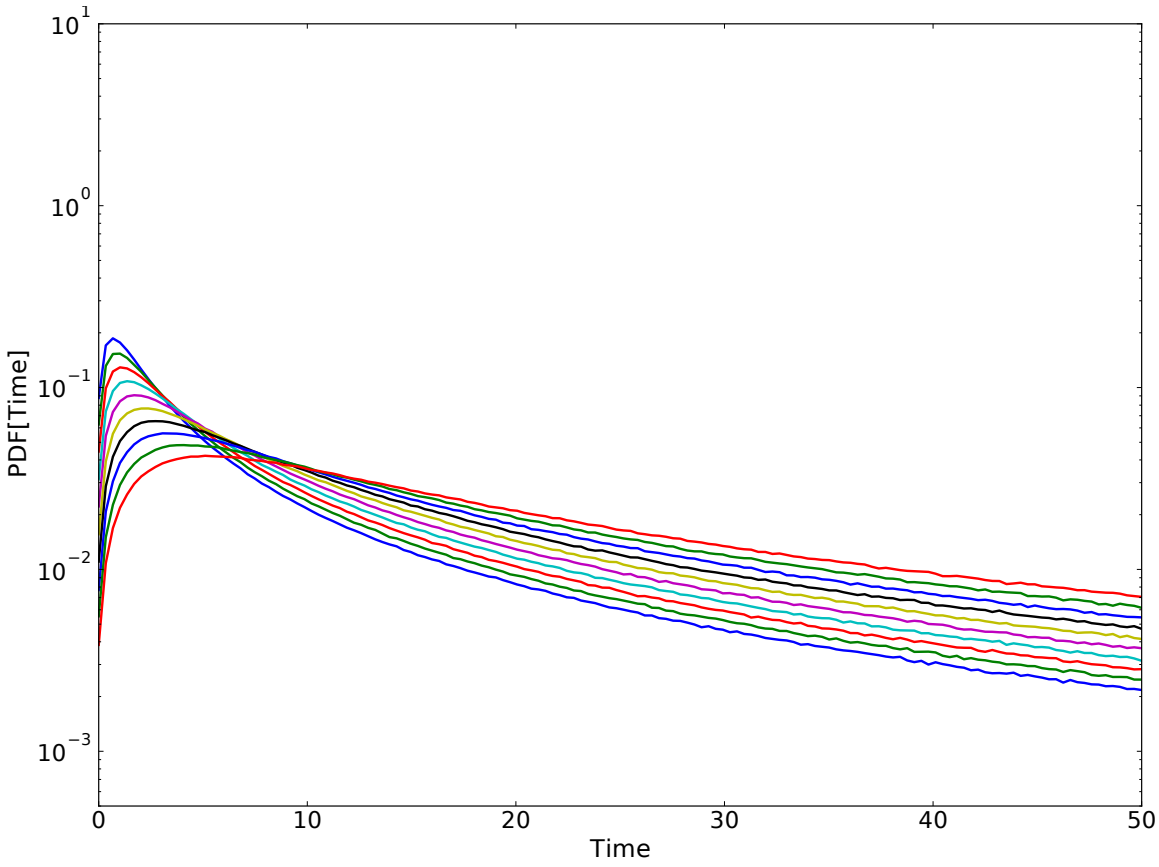


Figure S4: Genetic Toggle Switch phase zero inter-event time distribution for values of θ in between $.1$ and 10 (non-inclusive). 10^7 samples in each distribution.

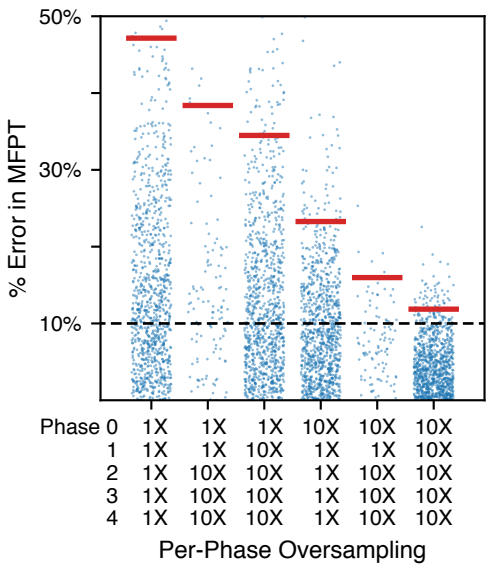


Figure S5: More results from simulations executed with various oversampling schemes. Aside from the use of 10X phase 0 oversampling instead of 20X, these simulations were equivalent to those presented in Fig 2.11 in the main paper. As can be seen in the right-hand column, 10X oversampling in every phase from 0 to 4 is almost, but not quite, enough to eliminate landscape error. The extent of oversampling in each separate scheme is indicated beneath each column of results. 1X sampling implies that standard number of FFPilot trajectories were sampled in that phase. For purposes of comparison, results from simulations run with no oversampling are included in the first column, and results from simulations run with 10X phase 0, 10X phase 1-4 oversampling are also shown. All simulations were of $\text{GTS}_{\theta=10}$ at an error goal of 10%.

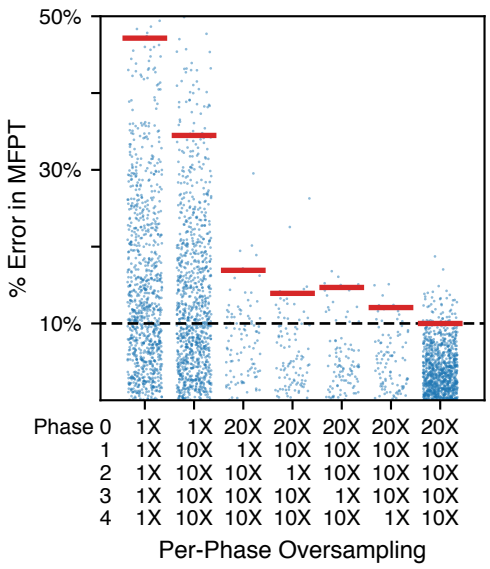


Figure S6: More results from simulations executed with various oversampling schemes. We wanted to investigate the effects of skipping oversampling in a single phase. Starting at the second column, results are shown from simulations in which we skipped oversampling in phase 0, then in the next column from simulations in which we skipped oversampling in phase 1, and so forth. The extent of oversampling in each separate scheme is indicated beneath each column of results. 1X sampling implies that standard number of FFPilot trajectories were sampled in that phase. For purposes of comparison, results from simulations run with no oversampling are included in the first column, and results from simulations run with 20X phase 0, 10X phase 1-4 oversampling (which is enough to eliminate landscape error) are also shown. All simulations were of $\text{GTS}_{\theta=10}$ at an error goal of 10%.

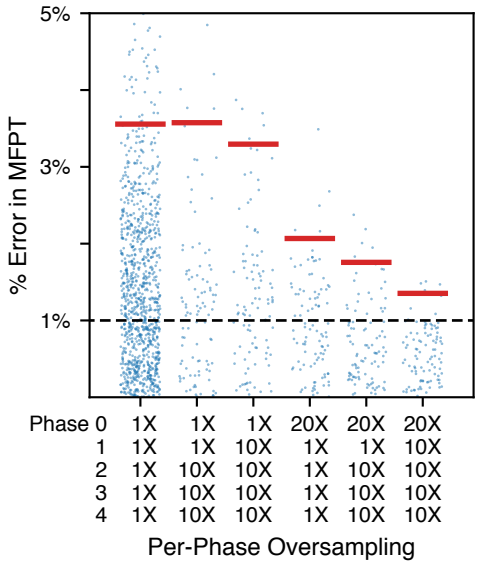


Figure S7: $MFPT$ percent errors from a large set of simulations executed with various oversampling schemes. The simulations are similar to those presented in Fig 2.11 from the main paper, except that all simulations were of $\text{GTS}_{\theta=10}$ at an error goal of 1% instead of 10%. All $MFPT$ percent errors are calculated relative to the $MFPT$ value estimated by a high accuracy DS simulation (.62% error goal).

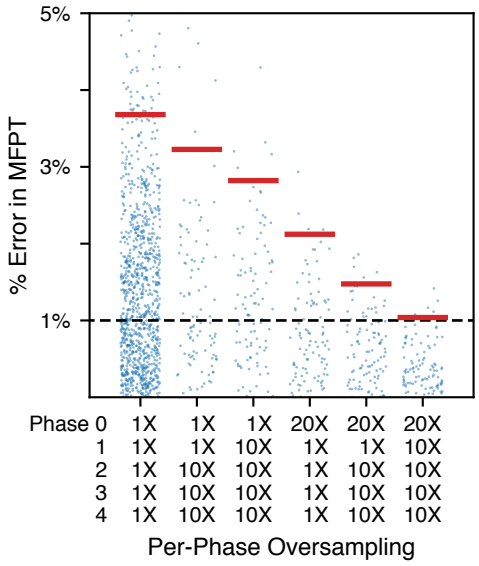


Figure S8: $MFPT$ percent errors from a large set of simulations executed with various oversampling schemes. The simulations are similar to those presented in Fig 2.11 from the main paper, except that all simulations were of $\text{GTS}_{\theta=10}$ at an error goal of 1% instead of 10%. All $MFPT$ percent errors are calculated relative to the $MFPT$ value estimated by a high accuracy FFPilot simulation (.1% error goal).

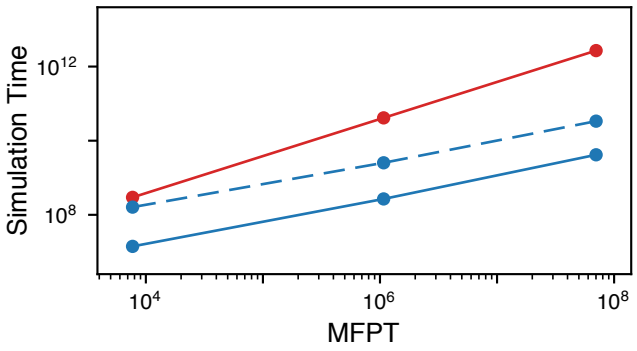


Figure S9: Simulation time vs $MFPT$ for several SRG models. The simulation time for both DS (red lines) and FFPilot (blue lines) are shown.

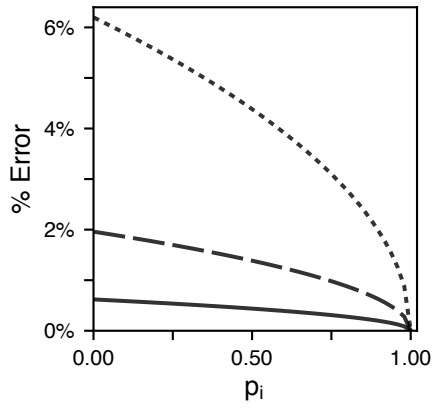


Figure S10: Maximum error (with respect to a single phase weight, 95% confidence) vs phase weight for a phase $i > 0$ when using the blind optimization method from the FFPilot pilot stage. Each of the lines shows the error for a different fixed value of the successful trajectory count, $n_i^s = n_{\text{pilot}}$. (solid line) $n_{\text{pilot}} = 10^5$, (dashed line) $n_{\text{pilot}} = 10^4$, (dotted line) $n_{\text{pilot}} = 10^3$.

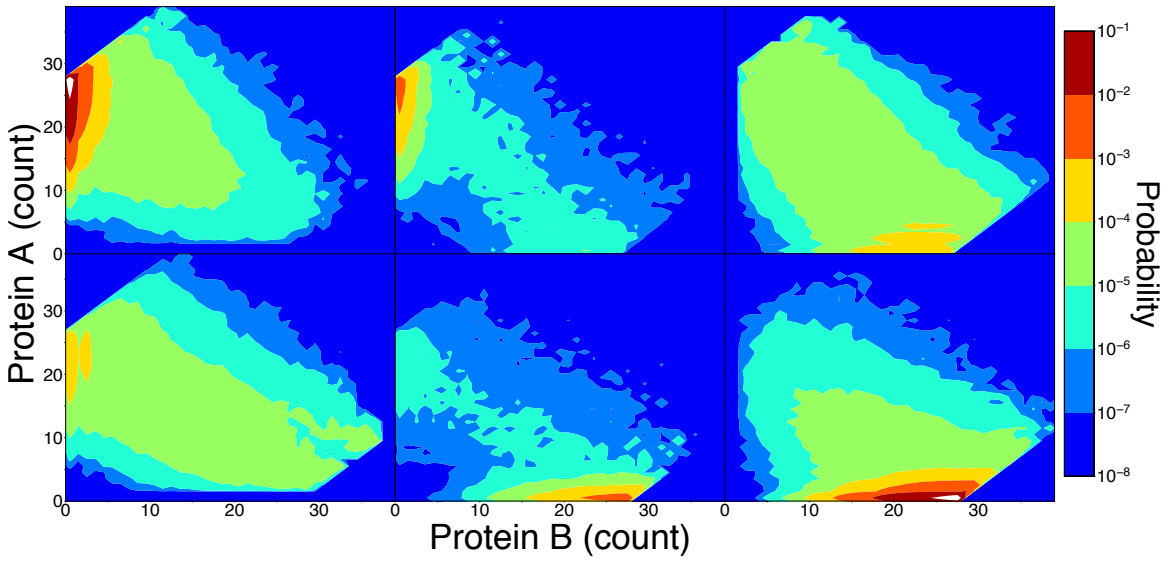


Figure S11: The state landscape of GTS$_{\theta=10}$, sliced by operator occupancy. The lefthand column shows the slice of the landscape in which an A dimer is bound to the operator DNA, the middle column shows the slice in which the operator is unbound, and the righthand column shows the slice in which the operator is bound to a B dimer. The top row is from a simulation of GTS$_{\theta=10}$ in which it switched $\mathcal{A} \rightarrow \mathcal{B}$, and the bottom row is from a simulation in which it switch $btoa$. Complex, high dimensional, and reasonably accurate landscapes can be produced in a straightforward fashion from relatively low cost FFPilot simulations. The 3 landscapes in each row represent the output of a single FFPilot simulation run to a 10% error goal, which ran to completion on a laptop in 5 minutes. The landscapes were analyzed and plotted using LMA, an analysis package written in Python and designed to work alongside the Lattice Microbes simulation suite.

2.S.4 Supplemental tables

Table S1: Deterministic ordinary differential equation model of the GTS.

Deterministic Genetic Toggle Switch Equations
$$\frac{dA}{dt} = -10A^2 + 10A_2 + \theta\left(-\frac{A}{4} + OA_2 + O\right)$$
$$\frac{dA_2}{dt} = 5A^2 + OA_2 - 5A_2\left(O + 1\right)$$
$$\frac{dOA_2}{dt} = -OA_2 + 5A_2 \cdot O$$
$$\frac{dB}{dt} = -10B^2 + 10B_2 + \theta\left(-\frac{B}{4} + OB_2 + O\right)$$
$$\frac{dB_2}{dt} = 5B^2 + OB_2 - 5B_2\left(O + 1\right)$$
$$\frac{dOB_2}{dt} = -OB_2 + 5B_2 \cdot O$$
$$\frac{dO}{dt} = 1 - OA_2 - OB_2$$

References

[1] Rosalind J Allen, Patrick B Warren, and Pieter Rein ten Wolde. “Sampling rare switching events in biochemical networks.” In: *Phys Rev Lett* 94.1 (2005), p. 018104.

[2] Daniel T Gillespie. “On the calculation of mean first passage times for simple random walks”. In: *J Chem Phys* 74.9 (1981), pp. 5295–5299.

[3] Chantal Valeriani, Rosalind J Allen, Marco J Morelli, Daan Frenkel, and Pieter Rein ten Wolde. “Computing stationary distributions in equilibrium and nonequilibrium systems with forward flux sampling.” In: *J Chem Phys* 127.11 (2007), p. 114109.

[4] Nils B Becker and Pieter Rein ten Wolde. “Rare switching events in non-stationary systems.” In: *J Chem Phys* 136.17 (2012), p. 174119.

[5] David Olive. *Statistical Theory and Inference*. Cham: Springer, 2014.

[6] Sheldon M Ross. *Introductory Statistics*. Academic Press, 2010.

[7] L Carlitz. “The inverse of the error function”. In: *Pacific Journal of Mathematics* (1963).

[8] G W Oehlert. “A note on the delta method”. In: *The American Statistician* (1992).

[9] Larry Wasserman. *All of Statistics*. A Concise Course in Statistical Inference. New York, NY: Springer Science & Business Media, 2013.

[10] Rosalind J Allen, Daan Frenkel, and Pieter Rein ten Wolde. “Forward flux sampling-type schemes for simulating rare events: efficiency analysis.” In: *J Chem Phys* 124.19 (2006), p. 194111.

[11] K F Riley, M P Hobson, and S J Bence. *Mathematical Methods for Physics and Engineering*. A Comprehensive Guide. Cambridge University Press, 2006.

[12] Ernesto E Borrero and Fernando A Escobedo. “Optimizing the sampling and staging for simulations of rare events via forward flux sampling schemes”. In: *J Chem Phys* 129.2 (2008), pp. 024115–024117.

[13] Nils B Becker, Rosalind J Allen, and Pieter Rein ten Wolde. “Non-stationary forward flux sampling.” In: *J Chem Phys* 136.17 (2012), p. 174118.

[14] Elijah Roberts, Shay Be’er, Chris Bohrer, Rati Sharma, and Michael Assaf. “Dynamics of simple gene-network motifs subject to extrinsic fluctuations”. In: *Phys Rev E* 92.6 (2015), pp. 062717–14.

[15] T S Gardner, C R Cantor, and J J Collins. “Construction of a genetic toggle switch in Escherichia coli”. In: *Nature* 403.6767 (2000), pp. 339–342.

[16] Tommaso Biancalani and Michael Assaf. “Genetic Toggle Switch in the Absence of Cooperative Binding: Exact Results”. In: *arXiv.org* 20 (2015), p. 208101.

[17] Jan R Magnus and Heinz Neudecker. *Matrix Differential Calculus with Applications in Statistics and Econometrics*. University of Texas Press, 1999.

[18] L A Goodman. “The variance of the product of K random variables”. In: *Journal of the American Statistical Association* (1962).

[19] Alexandr A Borovkov. *Probability Theory*. Universitext. London: Springer Science & Business Media, 2013.

Chapter 3

Discussion and conclusion

The epigenetic landscapes of complex genetic regulation networks can be mapped using stochastic simulation. The presence of rare events in these regulatory networks makes the traditional stochastic methods non-practical (in terms of computation time). Our theoretical analysis shows for the first time that a complete FFS simulation, including phase 0, can indeed be used to significantly speed up stochastic simulation of complex systems with rare events. We also show that there are significant sources of error in the FFS approach that have not been fully considered by the existing analyses.

The FFPilot method is an enhanced sampling method that controls error in an optimal way. For simulations of simple enough systems, this error control is exact and precise. For any system with a 1D landscape, FFPilot will produce results with a consistent, predictable, user-set upper bound on the margin of error. For multi-dimensional systems without rough landscapes, error is still controlled, but not precisely. For these more complex systems there is an anomalous extra error that is not fully accounted for by the theory and algorithms that drive FFPilot. The reasons why are complicated, but can be summarized with one word: covariance. We established that correlations between the sets of states used in each phase to launch new trajectories are sufficient to explain the anomalous error.

It is possible to perform an alternative derivation of the simulation variance that

takes the phase-by-phase covariance into account. Instead of zeroing out the covariances in Eq 2.18, they can be left as-is. This will end up giving what is effectively an extra term in the definition of the margin of error given in Eq 2.21:

$$\zeta\left[\widehat{W},\alpha\right] = z_\alpha\sqrt{\sum_i\sum_j \frac{\sigma_{ij}}{w_i w_j\sqrt{n_i}\sqrt{n_j}}},$$

$$\zeta\left[\widehat{W},\alpha\right] = z_\alpha\sqrt{\sum_i \frac{V\left[\mathbb{w}_i\right]}{{w_i}^2 n_i} + \sum_i\sum_{j\neq i} \frac{\sigma_{ij}}{w_i w_j\sqrt{n_i}\sqrt{n_j}}}.$$

Initial investigation suggests that a simple closed form equation that optimizes the run counts n_i and n_j (like Eq 2.28 or Eq 2.31) cannot be found when the covariance term is included. However, it is still likely that a reasonable optimization scheme (probably involving some numerical minimization) can be developed. Additionally, it should be possible to estimate the phase-by-phase covariances σ_{ij} using some variant of the scheme currently used by FFPilot to estimate the phase zero variance $V\left[\mathbb{w}_0\right]$. Thus, it is likely that an FFPilot version 2 could be developed that would fully take the covariances into account.

In summary, we show that it is possible to get speedups on the order of 100x when simulating genetic regulatory networks by using enhanced sampling methods. As the enhanced sampling methods, and the theory behind them, are further refined, it should be possible to get the fullest possible speedup in a completely automated way, one that is no more complex to use than standard stochastic simulation methods.

Tutorial - enhanced sampling using Lattice Microbes and FFPilot

Table of Contents

Chapter 1

FFPilot simulation: conceptual overview

1.1 How to use FFPilot simulation

1.1.1 Use cases

FFPilot can be used to greatly accelerate the simulation of a wide variety of systems. The primary use case for FFPilot is the simulation of systems with two (or more) well defined meta-stable states. With the correct setup, FFPilot will ”ratchet” the simulation from one state to the other, yielding a great deal of information about the transition process and the probability landscape around and in-between the states.

In general, FFPilot should have a speed advantage over standard replicate simulation for any system that has a rough probability landscape with at least some large energetic barriers (equivalently, for any system encompassing rare events).

1.1.2 Non-use cases

In theory, FFPilot simulation should be faster (in terms of internal simulation time) than replicate sampling in all cases. However, for certain types of systems the benefit of using FFPilot is expected to be minor:

- **Systems with flat landscapes/no rare events**
  Systems in which the probability landscape is nearly flat, in that it lacks any significant barriers. In other words, systems with 0 fixed points.
- **One-state systems**
  Systems that spend virtually all of their time in the neighborhood of a single point in their state space. In other words, systems with 1 fixed point.

For these types of systems replicate simulation will likely be faster (in terms of wall clock time) than FFPilot, due to various optimizations in the current version of Lattice Microbes. Thus, it is better to avoid the extra setup involved with using FFPilot when modeling these types of systems.

1.2 FFPilot specific concepts

FFPilot simulation is straightforward to set up, given that you at least know the rough locations (in terms of your system's state space) of the steady states of interest.

1.2.1 Order parameters

FFPilot uses an order parameter to distinguish one stable state from another, and to measure progress along a transition path. If FFPilot is thought of as driving a system from one state to another, then the order parameter defines the direction in which that driving occurs.

An order parameter is a function $\mathcal{O} : \left\{\mathbb{N}_{\geq 0}^{s},\ \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^{O}\right\}$ of the species counts and time of a system:

$$\mathcal{O}\left(s_0\,,\ \cdots\,,\ s_s\,,\ t\right) = \{o_0\,,\ \cdots\,,\ o_O\}$$

An order parameters can thought of as a quantitative, mathematically defined version of a reaction coordinate.

1.2.2 Tilings

A tiling is a set of bins that spans the state space of a system. A tiling is defined in terms of an order parameter, a set of edges, and one or more initial basins (from which to start trajectories during phase zero). Multiple basins can be defined in order to carry out both forward and reverse simulation in a single run of Lattice Microbes ES (LMES).

Tilings can be defined in terms of a set of bin edges (also sometimes referred to as a set of interfaces). An FFPilot simulation will have as many

- The edges are held in a 1D array called (appropriately enough) Edges. - The initial basins are held in an N x D array called Basins, where N is the count of basins and D is the total number of unique chemical species in your system. - Each row in Basins represents a different basin, and each column in Basins holds the count of one particular chemical species in the model. - In other words, a basin is a single coordinate in the state space of the model. - Each initial basin should either be in front of the zeroth edge or behind the last edge. - Mathemtically, this means that one of

$$\mathcal{O}(\text{basin}) < \mathcal{O}(\text{Edges}[0])$$

or

$$\mathcal{O}(\text{basin}) > \mathcal{O}(\text{Edges}[-1])$$

should be true, where $\mathcal{O}$ is the order parameter of the tiling.

The rows in the 'Basins' array determine where trajectories are started from during phase zero. A separate pilot and production stage (ie a whole separate FFPilot simulation) is run for each row in 'Basins'. Thus, setting two different basins is a convenient way to run the simulations required to calculate the MFPT of both the forward switch (low A → high A) and the reverse switch (high A → low A) of the self regulating gene.

Chapter 2

Calculating values of interest from FFPilot output: theory

2.1 Mean first passage time

The mean first passage time $MFPT_i$ to each edge i of the tiling is calculated in terms of the weights w_i as:

$$MFPT_i = \begin{cases} w_0 & \text{for } i = 0 \\ \frac{w_0}{\prod_{j=1}^{i} w_i} & \text{for } i > 0 \end{cases} \tag{2.1}$$

Thus, the overall $MFPT$ from the low A state to the high A state can be calculated as:

$$MFPT = MFPT_N = \frac{w_0}{\prod_{j=1}^{N} w_i} \tag{2.2}$$

where N is the index of the final edge.

In terms of the phase weights w_i, the formula for the mean first passage time to the jth edge is:

$$\frac{w_0}{\prod_{i=1}^{j} w_i}$$

2.2 Landscapes produced by FFPilot

One of the primary goals of stochastic simulation of biological systems is to calculate the probability landscapes of those systems. For nonequilibrium steady-state systems, calculating the landscape is equivalent to solving the system's master equation. From it you can obtain complete information about the occupancies and fluxes of the system.

When working with FFPilot simulation output, the calculation of the complete unbiased probability landscape of a system can be thought of as the recursive process of building up larger histograms (in terms of their information content) from smaller ones. Of course the final, complete landscape is the one of primary interest, but the phase, stage, and transition region landscapes that you build in

the process can be helpful for investigating the fine details of a system or of the simulation process itself.

2.2.1 FFPilot phase landscapes

The first step in assembling the complete landscape hist is also the simplest. You just take the state samples collected during the FFPilot simulation and group them by phase. You then just bin them together into sparse histograms, one per phase run during a production stage.

2.2.2 FFPilot stage landscapes

All of the phase $i > 0$ hists (the phase 0 hists are dealt with later) are separated into groups based on their originating stage. Next, the phase hists in each group are combined via a simple weighted sum of their values. The weights, which we'll call the landscape weights $\mathbb{I}_i$, are calculated by the following formula:

$$\mathbb{I}_i = \begin{cases} \frac{1.0}{n_1} & \text{for } i = 1 \\ \frac{\prod_{j=1}^{i-1} w_j}{n_i} & \text{for } i > 1 \end{cases} \tag{2.3}$$

where w_i is the phase i weight and n_i is the total count of trajectories run in phase i*.

The indices in eqrefeq:landweight are a bit confusing, so to give you a concrete example say that you ran a simulation that had 4 phases per stage. Phase 0 doesn't get a landscape weight, so your three $\mathbb{I}_i$ values would be:

$$\mathbb{I}_1 = \frac{1.0}{n_1}$$
$$\mathbb{I}_2 = \frac{w_1}{n_2}$$
$$\mathbb{I}_3 = \frac{w_1 w_2}{n_3}$$

Due to the boundary conditions imposed in phases $i > 0$, the stage hist will only cover the region of state space that was spanned by the tiling† used to setup the FFPilot simulation. We will refer to this span as the transition region, since by construction it will lie in-between two stable fixed points. The ith stage hist can be thought of as the conditional probability landscape of the transition region, given that the ith basin was the most recently visited. In other words, if a trajectory is in (or was most recently in) basin i, the ith stage hist will be effectively the instantaneous probability landscape for that trajectory.

*Eq 2.3 only applies if you're taking state samples at a constant interval. Though discussion of a variable sampling time step is outside of the scope of this tutorial, the landscape weight formula can be modified [1] to allow for one.

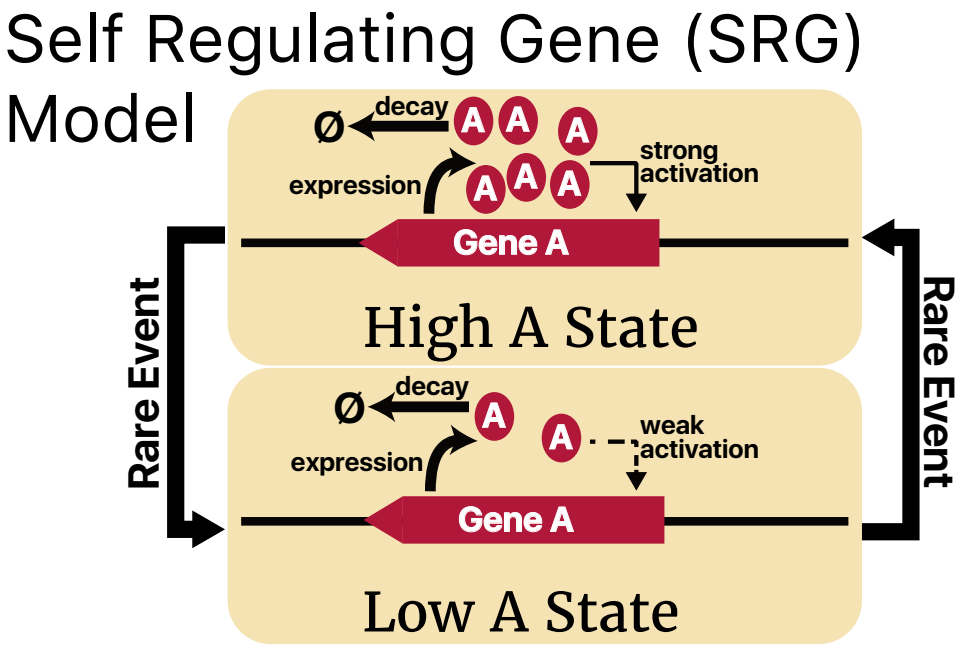
†Technically, what we mean by spanned region is all points x in state space such that $\lambda_0 <= \mathcal{O}(x) < \lambda_N$

Chapter 3

Using FFPilot to calculate mean first passage time (MFPT): self regulating gene

3.1 Overview

In this section I'll go over how to use FFPilot to calculate the mean first passage time (MFPT) of the self-regulating gene (SRG). SRG consists of a single protein, which I'll call protein A. Protein A participates in a positive feedback loop in which it upregulates its own production. Additionally, it experiences constant degradation. These properties make SRG bistable, and it will stochastically switch from one state, with low levels of protein A, to another, with high levels of protein, and back again over time.



Species	Description
A	protein

Reactions	Rates	Description
$\varnothing \longrightarrow \mathrm{A}$	$k_{low} + (k_{high} - k_{low}) \frac{\mathrm{A}^h}{k_{50}^h + \mathrm{A}^h}$	expression
$\mathrm{A} \longrightarrow \varnothing$	$\beta \mathrm{A}$	decay

Starting from a SRG model file included with this tutorial, `notebooks/self_regulating_gene.sbml`, I'll show you how to set up the simulation input file, execute the simulation, and analyze the output. If you execute the example commands and code exactly as written, you should end up with a new directory `notebooks/srg_mfpt/data` that contains `srg.lm`, the simulation input file, and `srg_out.sfile`, the simulation output file. Pre-made example input and output files can be found in `notebooks/srg_mfpt/data_worked`.

3.2 Input file setup

The input file for an FFPilot simulation is the same as for standard (*i.e.* direct sampling) simulation. However, some extra information is required, in the form of an order parameter and a tiling.

3.2.1 Convert .sbml -> .lm

Models of biochemical systems are often distributed in the SBML format. These can be converted to the `.lm` format required by Lattice Microbes using the `lm_sbml_import` utility.

Open a terminal and `cd` to the `notebooks/srg_mfpt` directory. You will then execute the following commands:

```
user@host:srg_mfpt$ mkdir -p data
user@host:srg_mfpt$ lm_sbml_import data/srg.lm ../self_regulating_gene.sbml
```

The first command creates a separate `notebooks/srg_mfpt/data` directory which we will use to hold any simulation input/output that we produce during this section of the tutorial. The last command converts `notebooks/self_regulating_gene.sbml` file using `lm_sbml_import` and then places the resulting `.lm` file at `notebooks/srg_mfpt/data/srg.lm`.

3.2.2 Add an order parameter

If fed into LMES in its current form, `srg.lm` could be used to run a standard replicate simulation. However, in order to use `srg.lm` as input for an FFPilot simulation, we'll have to first open it up and add some extra information: an order parameter and a tiling. We'll start with the order parameter.

Internally, `.lm` files are based on the `hdf5` format. This means that all of the existing `hdf5` software tools can directly interact with `srg.lm`, just the same as for any `hdf5` file. We'll be using two of these `hdf5` tools in this tutorial:

`h5py`
: A Python package that can open and/or modify `hdf5` files from within Python code. We'll be using `h5py` to modify the input file/add the extra information.

`h5dump`
: A command line tool that can be used to dump the contents of a `hdf5` file in plain text to the `stdout` of a terminal. We'll be using `h5dump` to show the correct format for the extra input required by FFPilot.

The order parameter that we will use with SRG, which I will call $\mathbf{A}$, will just be the count of the single chemical species in the system, protein A. Order parameter $\mathbf{A}$ is a linear combination of species counts and can thus be represented in LMES as a linear order parameter. The following Python script will open up `srg.lm` and add the definition of $\mathbf{A}$ in the required format:

```python
import h5py

with h5py.File('data/srg.lm') as f:
    # remove any existing OrderParameters group
    if 'OrderParameters' in f.keys():
        del f['OrderParameters']

    # create the OrderParameters group
    oparams = f.create_group('OrderParameters')

    # add the subgroup for order parameter 0
    oparam0 = oparams.create_group('0')

    # set this order parameter's ID to 0, and its Type to 0
    oparam0.attrs['ID'] = 0
    oparam0.attrs['Type'] = 0
```

```
    # add the SpeciesIDs. This array should always be of type `int`
    speciesIDs = np.array([0], dtype=int)
    oparam0.create_dataset('SpeciesIDs', data=speciesIDs)

    # add the SpeciesCoefficients. This array should always be of type `float`
    speciesCoefficients = np.array([1.0], dtype=float)
    oparam0.create_dataset('SpeciesCoefficients', data=speciesCoefficients)
```

The above script adds a new group, `OrderParameters`, to `srg.lm`, the contents of which we can inspect using `h5dump` by opening a terminal, `cd`-ing to `notebooks/srg_mfpt`, and then running the following command:

```
user@host:srg_mfpt$ h5dump --group=OrderParameters data/srg.lm
```

The dump will show that the `OrderParameters` group contains one subgroup, `0`:

```
HDF5 "data/srg.lm" {
GROUP "OrderParameters" {
   GROUP "0" {
```

Group `OrderParameters/0` contains the definition of $\mathbf{A}$. Two attributes are set on group `0`, `ID` and `Type`:

```
      ATTRIBUTE "ID" {
         DATATYPE  H5T_STD_I64LE
         DATASPACE  SCALAR
         DATA {
         (0): 0
         }
      }
      ATTRIBUTE "Type" {
         DATATYPE  H5T_STD_I64LE
         DATASPACE  SCALAR
         DATA {
         (0): 0
         }
      }
```

These attributes, which are required in every order parameter definition, do the following:

`ID` This attribute (which should match the name of the `OrderParameters` subgroup on which it is set) is the name by which LMES will refer to $\mathbf{A}$ internally.

`Type`
This attribute tells LMES what kind of order parameter this is, and how it should interpret the order parameter's definition when calculating its value. A `Type` of `0` tells LMES that $\mathbf{A}$ is a linear order parameter.

Inside of group `0` are two 1D arrays (by convention, `hdf5` tools refer to arrays as datasets), `SpeciesIDs` and `SpeciesCoefficients`:

```
      DATASET "SpeciesCoefficients" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
         DATA {
         (0): 1
         }
      }
      DATASET "SpeciesIDs" {
         DATATYPE  H5T_STD_I64LE
         DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
         DATA {
         (0): 0
         }
      }
```

LMES uses the values in these arrays to define the formula for the value of an order parameter. For any linear order parameter $\mathcal{O}$, LMES will use the following formula to calculate its value:

$$\mathcal{O} = \sum_{i=0}^{\texttt{len(SpeciesIDs)}-1} \texttt{Counts[SpeciesIDs[i]]} * \texttt{SpeciesCoefficients[i]} \tag{3.1}$$

where `Counts` is a 1D array containing the present count of each species in the system being simulated. Eq 3.1 can be simplified considerably for $\mathbf{A}$:

$$\mathbf{A} = \texttt{Counts[SpeciesIDs[0]]} * \texttt{SpeciesCoefficients[0]}$$
$$\mathbf{A} = \texttt{Counts[0]}$$

3.2.3 Add a tiling

Now we add a tiling to the self regulating gene. The tiling will be defined in terms of order parameter $\mathbf{A}$. In the previous section we added $\mathbf{A}$ to `srg.lm` and gave it an `ID` of `0`. The following Python script will open up `srg.lm` and add a new tiling with an `ID` of `0`:

```
import h5py

with h5py.File('data/srg.lm') as f:
    # remove any existing Tilings group
    if 'Tilings' in f.keys():
        del f['Tilings']

    # create the Tilings group
    tilings = f.create_group('Tilings')

    # add the subgroup for tiling 0
    tiling0 = tilings.create_group('0')
```

```
    # set this tiling's ID to 0, its OrderParameterID to 0, and its Type to 0
    tiling0.attrs['ID'] = 0
    tiling0.attrs['OrderParameterID'] = 0
    tiling0.attrs['Type'] = 0

    # add the Basins. There is 1 chemical species in this model, and we are
    # going to set 2 initial basins, so Basins will be a 2 x 1 array
    basins = np.zeros((2,1), dtype=int)
    basins[0,0] = 10
    basins[1,0] = 160
    tiling0.create_dataset('Basins', data=basins)

    # add the Edges
    edges = np.linspace(23.0, 150.0, num=13)
    tiling0.create_dataset('Edges', data=edges)
```

Once the script has run, we can inspect the contents of the newly added `Tilings` group in `srg.lm` by opening a terminal, `cd` -ing to `notebooks/srg_mfpt`, and then running the following command:

```
user@host:srg_mfpt$ h5dump --group=Tilings data/srg.lm
```

The dump shows one subgroup, `0`, in the `Tilings` group:

```
HDF5 "data/srg.lm" {
GROUP "Tilings" {
   GROUP "0" {
```

Group `Tilings/0` has three required attributes set on it:

```
      ATTRIBUTE "ID" {
         DATATYPE  H5T_STD_I64LE
         DATASPACE  SCALAR
         DATA {
         (0): 0
         }
      }
      ATTRIBUTE "OrderParameterID" {
         DATATYPE  H5T_STD_I64LE
         DATASPACE  SCALAR
         DATA {
         (0): 0
         }
      }
      ATTRIBUTE "Type" {
         DATATYPE  H5T_STD_I64LE
         DATASPACE  SCALAR
         DATA {
         (0): 0
```

```
      }
   }
```

the description of which are as follows:

`ID` This attribute (which should match the name of the `Tiling` subgroup on which it is set) is the name by which LMES will refer to this tiling internally.

`Type`
A tiling `Type` of `0` tells LMES that this is a 1D linear tiling*.

`OrderParameterID`
The `ID` of the order parameter (in this case, **A**) that will be used in the definition of this tiling. Specifically, the placement of the edges that separate the tiles/bins of this tiling will be specified in terms of the tiling's order parameter.

Inside of group `Tilings/0` are a 2D array, `Basins`, and a 1D array, `Edges`:

```
   DATASET "Basins" {
      DATATYPE  H5T_STD_I64LE
      DATASPACE  SIMPLE { ( 2, 1 ) / ( 2, 1 ) }
      DATA {
      (0,0): 10,
      (1,0): 160
      }
   }
   DATASET "Edges" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 13 ) / ( 13 ) }
      DATA {
      (0): 23, 33.5833, 44.1667, 54.75, 65.3333, 75.9167, 86.5, 97.0833,
      (8): 107.667, 118.25, 128.833, 139.417, 150
      }
   }
```

These arrays are used in the following way:

`Edges`
Each entry in this array holds a single coordinate given in terms of the tiling's order parameter. LMES builds up a tiling by placing edges at these coordinates. In turn, the tiles/bins themselves are defined as the space in between any two adjacent edges. The size of the `Edges` array is directly related to the number of separate phases run during an FFPilot simulation. During each stage there will be exactly as many phases executed as there are edges in the tiling used to set up the stage.

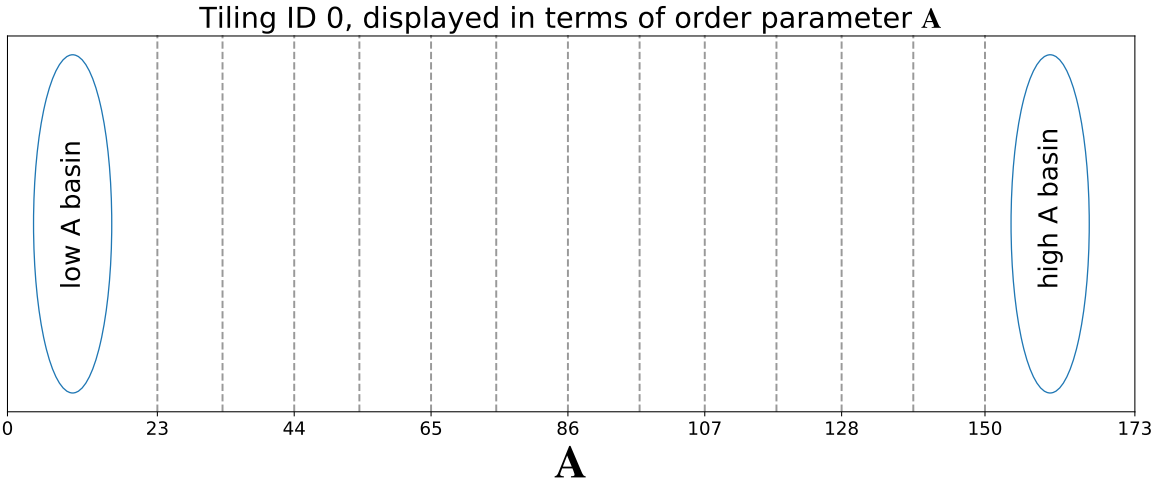
`Basins`
An N x D array, where N is the count of basins and D is the total number of unique chemical species in your system. The basins (which are defined in terms of a model's complete state space) are the points in the state space of a system from which the initial trajectories of each

*1D linear is the only kind of tiling currently implemented in LMES. Thus, for now, this is just a placeholder value that should always be set to 0. In the future, more complex tilings, such as ones based on the Voronoi tessellation [2], may be implemented.

stage (*i.e.* the phase zero trajectories) will be launched. Each row in `Basins` is treated as a separate basin.

The rows in the `Basins` array determine where trajectories are started from during phase zero. One complete self-contained FFPilot simulation (*i.e.* a pilot stage following by a production stage) will be executed for each basin specified in the simulation input's tiling. Thus, setting two different basins is a convenient way to run the simulations required to calculate the MFPT of both the forward switch (low A → high A) and the reverse switch (high A → low A) of the SRG model.

The Python script we used to create our tiling will have set up two basins. The initial basin is at 10 counts of protein A, corresponding to the low A state of SRG. This is located in front of the zeroth edge of tiling (which sits at 23 counts of A). The other basin is at 160 counts of protein A, corresponding to the high A state of SRG. This basin is located behind the last edge of the tiling (which sits at 150 counts of A). The following figure gives a sense of what the tiling actually looks like:



3.2.4 (advanced) FFPilot batch mode (multiple tilings)

In normal usage, the input file for an FFPilot simulation should have exactly one tiling. However, FFPilot simulations can be run in large batches by specifying multiple tilings. In this case, each separate tiling will be treated as if it describes a separate simulation. In other words, an FFPilot simulation run using an input that contains M separate tilings is equivalent to running a batch of M separate FFPilot simulations.

NB: when creating an input file to use with FFPilot batch mode, each tiling should have its own unique `ID`.

3.2.5 Customize FFPilot simulation options

FFPilot is designed to require as little user input as possible. All of the simulation options/parameters that FFPilot uses have reasonable (and, in some cases, adaptive) default values. On the other hand, FFPilot supports a number of options that can be used to control many different aspects of the simulation. In `.lm` input files, options are set as attributes on the top level `Parameters` group.

NB: `lmes` only recognizes options set with values of type `str`. All non-string options (*i.e.* `int`, `bool`, etc.) should be converted to `str` before they are set.

We'll set up some options for our FFPilot SRG simulation using the following python script (which contains several examples of `str` conversion):

```
import h5py

with h5py.File('data/srg.lm') as f:
    # NB: convert all option values to str before setting them

    # turn on output of pilot stage data
    f['Parameters'].attrs['ffluxPilotOutput'] = str(True)

    # set a target error goal for the simulation as a whole. Defaults to .05
    f['Parameters'].attrs['errorGoal'] = str(.10)
```

In the above script, we set three options:

`ffluxPilotOutput`
: `bool`, defaults to `False`. Turns on output of records from any pilot stages run during the simulation. Normally only simulation output from the production stages is saved, and the data from every pilot stage is discarded once its corresponding production stage has been set up. When this flag is set to `True`, however, output from both the pilot and the production stages is saved.

`errorGoal`
: Percentage, specified as a `float` in the range `(0-1.0]`, defaults to `.05`. Sets a goal for the (percent) error in the overall basin-to-basin MFPT calculated by each production stage. Based on the outcome of the preceding pilot stage, FFPilot automatically determines how many trajectories to launch in each phase of a production stage in order to achieve a sampling error at or below the set error goal. Set `errorGoal` to `.01` for a high accuracy simulation, or set it to `.10`+ for a relatively fast simulation.

3.3 Running the simulation

Now that the FFPilot input is prepared, you can execute the simulation by opening a terminal, `cd`-ing to `notebooks/srg_mfpt`, and entering the following command:

```
user@host:srg_mfpt$ lmes -fflux -f data/srg.lm
```

As shown above, running an FFPilot simulation requires two flags to be passed to the LMES executable:

`-ffpilot`
: Tells LMES to run in FFPilot mode. Without this flag a default replicate simulation would be run instead.

`-f <input-path>`
: Tells LMES that the simulation input file can be found at `<input-path>`.

The path at which the simulation output will be saved is automatically determined from the input path. In this case, that means the simulation output will be saved to `data/srg_out.sfile`. Alternatively, you can explicitly set the simulation output path by specifying the `-fo <output-path>` flag on the command line when you execute LMES.

Since this simulation is set up using a relatively loose 10% error goal, it should run to completion fairly quickly. When run on my laptop (3.1 GHz, 4 CPU cores), the simulation finishes in about 30 seconds.

3.3.1 (advanced) Preserving the simulation log

Normally, the `lmes` simulation log gets dumped directly to the terminal as the simulation runs. You can instead preserve the log for later perusal by running `lmes` with a slightly modified command:

```
user@host:srg_mfpt$ lmes -fflux -f data/srg.lm > data/srg.log
```

where we've saved the simulation log to `notebooks/srg_mfpt/data/srg.log` using `>`, the Linux redirection operator.

One downside of redirecting the log is that it will prevent any progress messages from getting printed out to the terminal, making it hard to tell if the simulation is running correctly. One workaround is to monitor the log file in real time using the `watch` command while the simulation adds to it:

```
user@host:srg_mfpt$ watch tail -n 25 data/srg.log
```

where the `-n` flag controls the number of lines that are displayed.

3.4 Analyzing the log

As an `lmes` simulation runs, an informative log is dumped to `stdout`. Aside from helping to monitor simulation progress, a large quantity of information about the outcome of an FFPilot simulation can be gleaned from the simulation log alone. For example, the MFPT estimate is printed out at the end of every stage, eliminating the need to dig around in the actual simulation output file for this value. As well, the log contains some unique metadata about the simulation, such as the total elapsed wall-clock time.

3.4.1 FFPilot simulation progress log messages

FFPilot simulations, like all `lmes` simulations, begin with a kickoff phase, during which input is parsed and all of the parallel processes are initialized and coordinated. Once the kickoff is complete

(usually only a few seconds when running `lmes` on a single computer), an FFPilot simulation begins in earnest with the start of the pilot stage, as signaled by the following line getting printed to the log:

```
Forward Flux stage 0 started (tiling_id: 0, basin_id: 0, stage_type: Pilot)
```

This type of progress message is printed out at the start of every simulation stage. Each comma-separated term in between the parentheses holds a piece of information about the current simulation stage:

`tiling_id`
: This gives the `ID` of the tiling used to set up the simulation stage. This matches the `ID` of the tiling we created earlier (see Sec 3.2.3), and confirms that it was used to set up this simulation stage.

`basin_id`
: This corresponds to the row index of `Basins` from which the basin species counts for this stage were taken. These basin species counts will be used to initialize phase 0 trajectories during this stage.

`stage_type`
: Tells you whether this is a pilot or a production stage. One of each kind of stage is run for each basin in each tiling present in the simulation input.

Another kind of progress message is printed out at the start of each phase zero:

```
Forward Flux phase 0 started (zeroth_edge: 23.00, \
    phase_limit: FORWARD_FLUXES >= 10000.00)
```

The terms in the parentheses give some information about how this phase will be executed:

`zeroth_edge`
: The coordinate of the tiling edge (given in terms of the tiling's order parameter) closest to the basin in which this phase's trajectories are initialized. Each time a phase zero trajectory crosses the `zeroth_edge` while traveling away from its starting basin, the simulation counts this as a single forward flux event. Additionally, the state that the trajectory was in when it fluxed forward is added to the dictionary of starting states that will be used to initialize trajectories during phase 1.

`phase_limit`
: Describes the condition that must be fulfilled in order for this phase to be considered complete. This particular `phase_limit` means that 10,000 forward flux events (summed across all phase zero trajectories) need to be observed before this phase will be terminated. Phase zero is always run using a `FORWARD_FLUXES` type `phase_limit`.

A slightly different progress message is printed out at the start of each phase $i > 0$:

```
Forward Flux phase 4 started (starting_edge: 54.75, goal_edge: 65.33, \
    phase_limit: FORWARD_FLUXES >= 10000.00)
```

where the values in the parentheses mean:

`starting_edge`
: The edge along which lies all of the starting states used to initialize trajectories during this stage.

`goal_edge`
: Once a trajectory has been launched from `starting_edge` it will continue to run until it either crosses the `zeroth_edge` (as specified in the most recent phase zero progress message) or this `goal_edge`. In either case the simulation is immediately terminated. However, if the trajectory was stopped because it crossed the `goal_edge`, the simulation counts this as a forward flux event. As well, the state the trajectory was in when it fluxed forward is added to the dictionary of starting states that will be used to initialize trajectories in the next phase.

The `phase_limit` term means the same thing in the phases $i > 0$ progress messages as it does in the phase zero progress message. However, unlike phase zero, the type of `phase_limit` used to run any given phase $i > 0$ varies depending on whether the phase is part of a pilot or a production stage. Essentially, the type of the `phase_limit` determines the phase's termination condition:

`FORWARD_FLUXES >= n`
: The type of `phase_limit` used during pilot stages. Each phase $i > 0$ will only be considered complete once `n` forward flux events (*i.e.* the count of trajectories that reached the `goal_edge`) have been observed, regardless of how many trajectories have been launched in total.

`TRAJECTORY_COUNT >= n`
: The type of `phase_limit` used during production stages. Each phase $i > 0$ is considered complete once the total number of trajectories that have been launched and run to termination is `n`.

3.4.2 Stage outcome log messages

Every time a pilot stage finishes, some of the highlights from its output are added to the log file:

```
The phase costs are:
[4.62, 0.151, 1.61, 2.41, 2.72, 2.09, 1.65, 1.22, 0.967, 1.01, 0.951, 1.25, 1.5]
The phase weight sample variances are:
[152, 0.018, 0.12, 0.228, 0.245, 0.155, 0.0665, 0.0211, 0.00473, 0.00296, \
    0.000693, 0.000495, 0.000396]
Conservative estimates of the phase weights are:
[4.62, 0.0179, 0.137, 0.345, 0.56, 0.8, 0.922, 0.974, 0.993, 0.995, 0.998, \
    0.998, 0.999]
Attempting to achieve error goal 0.05 (confidence level 0.95) \
    with the following optimized trajectory counts:
[32385, 497266, 51746, 23142, 14016, 9022, 5917, 3830, 2210, 1792, 1148, 1000, 1000]
```

phase costs
: The ith entry of this array holds the average cost, in terms of the internal simulation time, of collecting a single sample (*i.e.* observation of a forward flux event) during phase i.

variances
: The ith entry of this array holds the variance of the samples collected during phase i. The samples in question are used to estimate the phase weights at the end of each phase.

conservative weight estimates
: The ith entry of this array holds the estimate of phase weight i that is calculated at the end of each phase i. of the samples collected during phase i.

The values in these first 3 arrays are used to parameterize the FFPilot optimizing equation. The FFPilot optimizing is used to predict the optimal simulation plan (*i.e.* count of trajectories to launch in each phase) for the upcoming production stage. By optimal, I mean the simulation plan that will achieve the specified error goal with the least computational effort possible.

The reason why the weight estimates shown above are referred to as ”conservative” is that, following their initial calculation, they have each been run through a set of biased estimators. These biased estimators have been designed such that when the conservative weight estimates are used to parameterize the FFPilot optimizing equation, the optimizing equation is in turn slightly biased towards overestimating the number of trajectories required to achieve the error goal. This has the beneficial effect of helping to ensure overall simulation accuracy, albeit at a small cost in computational efficiency.

The last array in the pilot stage output log message contains the prediction of the FFPilot optimizing equation, given the values in the first 3 arrays:

optimized trajectory counts
: The trajectory counts in this array are used to set up the `phase_limits` during the pilot stage. In other words, each phase $i > 0$ of the production stage will be run until the count of trajectories that have run to completion reaches `optimized_trajectory_counts[i]`.

Each production stage also adds some the highlights from its results to the log once it finishes:

```
The phase costs are:
[4.72, 0.151, 1.58, 2.42, 2.71, 2.08, 1.66, 1.28, 1.02, 1.03, 0.974, 1.24, 1.49]
The phase weights are:
[4.72, 0.0186, 0.145, 0.353, 0.573, 0.814, 0.926, 0.977, 0.993, 0.998, 1, 1, 1]
The first passage times to each tile edge are:
[4.72, 254, 1.76e+03, 4.97e+03, 8.67e+03, 1.07e+04, 1.15e+04, 1.18e+04, 1.19e+04, \
    1.19e+04, 1.19e+04, 1.19e+04, 1.19e+04]
The overall first passage time from the starting basin to the last tile edge is:
1.19e+04
```

phase costs and weights
: Same as from the pilot stage, but produced by the more accurate production stage.

mean first passage times
: The ith entry in this array is the expected value of the time it takes to get from the simulation’s starting basin to the ith edge of the tiling.

overall mean first passage time
: The expected value of the time it takes to get from the simulation’s starting basin to the ending basin. Equivalent to the final entry in the MFPT array printed above. The basins can be thought of as the metastable states of the system, so the overall MFPT can also be thought of as the inverse of the switching rate between those states.

3.5 Browsing FFPilot output with dumpSFile

Output from an FFPilot simulation takes the form of an `SFile`. By default, given that your simulation input is named `<fname>.lm` the output is saved to `<fname>_-_out.sfile` (this can be overridden by setting the `-fo <output-name>` cmd line option when running LMES). `SFile` is a binary file format that holds information in the form of records. Each record begins with a short metadata section (record name, record type, data size in bytes) followed by a data section. For LMES output in the `SFile` format, the data section takes the form of a serialized protocol buffer message.

Although `SFiles` are not human readable, the `dumpSFile` tool provided by the `robertslab` Python package can be used to easily view the contents of any `SFile`. For each record in an `SFile`, `dumpSFile` converts the metadata into a human readable format, deserializes and formats the data portion, and then prints the results to `stdout`. The following command will dump the entire contents of `data/srg_out.sfile` to the terminal:

```
user@host:srg_mfpt$ dumpSFile data/srg_-_out.sfile
```

The output will look something like this:

```
data_worked/srg_-_out.sfile

/Simulations/0/Tilings/0/Basins/0/Stages/Pilot    protobuf:lm.fflux.io.FFluxStageOutputSummary    426
first_passage_times: [4.91, 270, 1.9e+03, 5.42e+03, 9.46e+03, 1.17e+04, 1.26e+04, 1.29e+04, 1.3e+04, 1.3e+04, 1.3e+04, 1.3e+04, 1.3e+04]
costs: [4.91, 0.151, 1.6, 2.41, 2.71, 2.08, 1.67, 1.23, 0.96, 1, 0.968, 1.22, 1.5]
weights: [4.91, 0.0182, 0.142, 0.351, 0.573, 0.811, 0.925, 0.976, 0.993, 0.998, 1, 1, 1]
edges: [23, 33.6, 44.2, 54.8, 65.3, 75.9, 86.5, 97.1, 108, 118, 129, 139, 150]

/Simulations/0/Tilings/0/Basins/0/Stages/Production    protobuf:lm.fflux.io.FFluxStageOutputSummary    426
first_passage_times: [4.76, 256, 1.82e+03, 5.1e+03, 8.81e+03, 1.08e+04, 1.19e+04, 1.21e+04, 1.21e+04, 1.22e+04, 1.22e+04, 1.22e+04, 1.22e+04]
costs: [4.76, 0.152, 1.59, 2.38, 2.7, 2.02, 1.78, 1.17, 0.989, 1.03, 1.05, 1.31, 1.51]
weights: [4.76, 0.0186, 0.141, 0.356, 0.579, 0.812, 0.909, 0.986, 0.995, 0.998, 0.998, 1, 0.999]
edges: [23, 33.6, 44.2, 54.8, 65.3, 75.9, 86.5, 97.1, 108, 118, 129, 139, 150]

/Simulations/0/Tilings/0/Basins/1/Stages/Pilot    protobuf:lm.fflux.io.FFluxStageOutputSummary    426
first_passage_times: [0.323, 3.81, 9.21, 18.5, 44.2, 111, 343, 1.08e+03, 2.74e+03, 5.65e+03, 7.99e+03, 8.99e+03, 9.16e+03]
costs: [0.323, 0.0343, 0.451, 0.967, 1.76, 2.51, 3.48, 4.28, 4.6, 4.92, 3.96, 2.89, 2.12]
weights: [0.323, 0.0849, 0.413, 0.497, 0.42, 0.397, 0.324, 0.317, 0.396, 0.485, 0.706, 0.889, 0.981]
edges: [150, 139, 129, 118, 108, 97.1, 86.5, 75.9, 65.3, 54.8, 44.2, 33.6, 23]

/Simulations/0/Tilings/0/Basins/1/Stages/Production    protobuf:lm.fflux.io.FFluxStageOutputSummary    426
first_passage_times: [0.308, 3.6, 8.71, 17.4, 40.5, 102, 322, 999, 2.54e+03, 5.25e+03, 7.38e+03, 8.42e+03, 8.58e+03]
costs: [0.308, 0.0341, 0.445, 0.956, 1.76, 2.51, 3.53, 4.28, 4.66, 4.94, 3.94, 3, 2.08]
weights: [0.308, 0.0856, 0.413, 0.502, 0.429, 0.396, 0.318, 0.323, 0.393, 0.484, 0.712, 0.876, 0.982]
edges: [150, 139, 129, 118, 108, 97.1, 86.5, 75.9, 65.3, 54.8, 44.2, 33.6, 23]
```

The reason why the short simulation we just ran produced so much data is that we had turned on extra output via the `ffluxPilotOutput` option[†].

For each record it finds, `dumpSFile` will first print out a line containing the record's metadata. This metadata has the following format:

```
record.name    record.dataType    record.dataSize
```

The data section of the record will then be printed out starting on the next line after the metadata.

[†]If the simulation had been run using the default output options, the output would contain only 2 records, the summaries from each of the production stages.

`dumpSFile` has many options that can help to tame the torrent of data. Here's a couple of the more useful ones:

`-i/--include <pattern0> <pattern1> ...`
: When this option is set, `dumpSFile` will attempt to match each regular expression `<patterni>` to each record. Records will only be printed out if every pattern matches either the record's `name` or its `dataType`. Otherwise, `dumpSFile` will skip the record. Patterns passed to `-i` are case insensitive.

  For example, the following `dumpSFile` command, which passes the terms "summary", "production", and "basins/1" to the `-i` flag:

```
user@host:srg_mfpt$ dumpSFile data/srg_-_out.sfile -i summary production basins/1
```

  The "basins/1" and "production" terms match to `record.name`, and the "summary" term matches to `record.dataType`. This limits output to a single stage summary record, the one from the production stage that was initialized in basin `ID 0`:

```
data_worked/srg_-_out.sfile

/Simulations/0/Tilings/0/Basins/1/Stages/Production    protobuf:lm.fflux.io.FFluxStageOutputSummary    426
first_passage_times: [0.308, 3.6, 8.71, 17.4, 40.5, 102, 322, 999, 2.54e+03, 5.25e+03, 7.38e+03, 8.42e+03, 8.58e+03]
costs: [0.308, 0.0341, 0.445, 0.956, 1.76, 2.51, 3.53, 4.28, 4.66, 4.94, 3.94, 3, 2.08]
weights: [0.308, 0.0856, 0.413, 0.502, 0.429, 0.396, 0.318, 0.323, 0.393, 0.484, 0.712, 0.876, 0.982]
edges: [150, 139, 129, 118, 108, 97.1, 86.5, 75.9, 65.3, 54.8, 44.2, 33.6, 23]
```

`-e/--exclude <pattern0> <pattern1> ...`
: The opposite of `-i/--include`. Records will be excluded if every `<patterni>` matches either its `name` or its `dataType`.

`-l/--list-only`
: When this flag is passed to `dumpSFile`, only the metadata line of each record is printed out. This can be very useful for getting a sense of the complete contents of an `SFile`. It can also be combined with the `--exclude/--include` options.

  For example, the following command:

```
user@host:srg_mfpt$ dumpSFile data/srg_-_out.sfile -l -i pilot
```

  will output only the metadata from the pilot stage records:

```
data_worked/srg_-_out.sfile

/Simulations/0/Tilings/0/Basins/0/Stages/Pilot    protobuf:lm.fflux.io.FFluxStageOutputSummary    426
/Simulations/0/Tilings/0/Basins/1/Stages/Pilot    protobuf:lm.fflux.io.FFluxStageOutputSummary    426
```

Run `dumpSFile --help` to get a detailed description of all of the options that `dumpSFile` supports.

3.6 Analyzing FFPilot output data using the robertslab.sfile Python package

3.6.1 Fetching and analyzing MFPT values in FFPilot simulation output

Although `dumpSFile` offers an easy way to browse the contents of FFPilot simulation output, it is less than ideal when used as a tool to perform detailed analysis. When performing any non-trivial calculations on FFPilot simulation output, the current recommended best practice is to use the `robertslab.sfile` Python package.

In this section I will go over how to retrieve and analyze data from an FFPilot output file using Python code. Specifically, I will walk you through the steps required to fetch the phase weights and MFPTs from the production stage summary records, and show you how to recalculate the MFPTs from the phase weights.

As can seen in the output of `dumpSFile` in the previous section, stage summary records each contain 4 arrays:

`edges`
: The positions of the edges in the tiling used to run the stage that produced this output.

`costs`
: The mean computational cost per-trajectory launched during each phase in this stage.

`first_passage_times`
: The expected value of the simulation time required to reach each tiling edge from the starting basin.

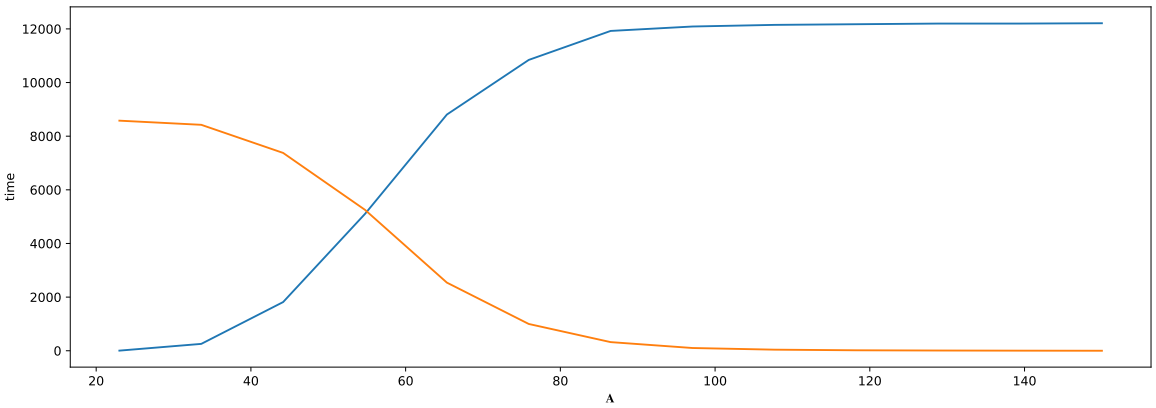
`weights`
: The phase weights.

For this exercise we want the stage summary records from the two production stages. These records can be fetched from `srg_out.sfile` with the following Python script:

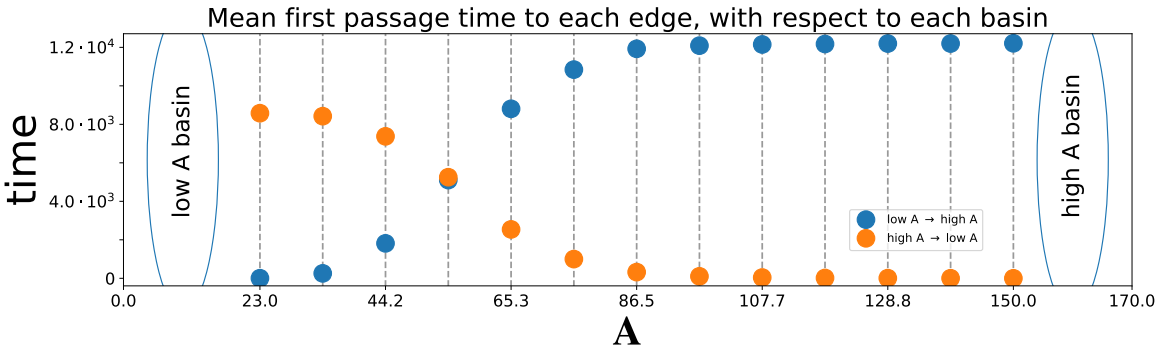
```
from robertslab.sfile import SFileProto

with SFileProto.open('data/srg_-_out.sfile') as f:
    summaries = list(f.msgs(include=('production', 'summary')))
```

A simple plot of the MFPT data can be made using the `matplotlib` Python plotting package:

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(12,4))
ax = fig.add_axes([0, 0, 1, 1])

for summary in summaries:
    ax.plot(summary.edges, summary.first_passage_times)

ax.set_xlabel('$\N{MATHEMATICAL BOLD CAPITAL ALPHA}$')
ax.set_ylabel('time')
pass
```



The plot can be made much nicer with a bit of configuration:



the details of which are outside of the scope of this tutorial. However, for the curious, the complete code used to produce the above figure can be found in `notebooks/src/plotTiling.py`, and a concrete example of its use is in the `notebooks/srg_mfpt.ipynb` notebook.

3.6.2 Calculating the MFPT to each tiling edge from the phase weights

FFPilot simulation calculates the MFPT to the ith edge of the tiling via a combination of the phase weights,

$$\frac{w_0}{\prod_{i=1}^{j} w_i}$$

(as per Eq 2.2). The value of this formula at all of the edges can be quickly calculated using a cumulative product. Given a sequence in which the zeroth term is w_0 and remaining terms are the inverse values each $w_{i>0}$, the ith element of the cumulative product of this sequence will be the expected value of the time required to reach the ith edge:

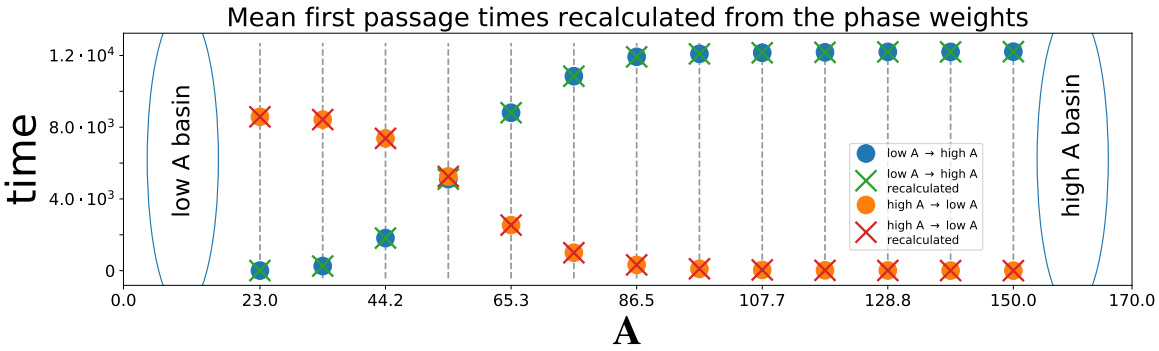
$$\text{cumprod}(\{w_0, \frac{1}{w_1}, \frac{1}{w_2}, \cdots, \frac{1}{w_N}\}) = \{w_0, \frac{w_0}{w_1}, \frac{w_0}{w_1 w_2}, \cdots, \frac{w_0}{\prod_{i=1}^{N} w_i}\}$$

The `sfile` package, in conjunction with `numpy`, can be used to easily apply this formula:

```
import numpy as np
from robertslab.sfile import SFileProto
```

```
with SFileProto.open('data/srg_-_out.sfile') as f:
    summaries = list(f.msgs(include=('production', 'summary')))

mfptRecalcs = []
for summary in summaries:
    # get a copy of the phase weights
    mfptRecalc = np.array(summary.weights)

    # invert all of the weights aside from the zeroth
    mfptRecalc[1:] = 1/mfptRecalc[1:]

    # now take the cumulative product accross all of the phase weights
    mfptRecalc[...] = np.cumprod(mfptRecalc)

    mfptRecalcs.append(mfptRecalc)
```

If you then plot the recalculated MFPT values against the MFPT values that were originally present in the output file you get:



demonstrating that they are indeed identical.

Chapter 4

Calculating the probability landscape of the self regulating gene (SRG) with FFPilot simulation

4.1 Overview

Now we move on to the calculation of a system's probability landscape. The landscape is a map which assigns a probability to every occupied state in a system's state space. Producing a landscape from FFPilot output required a much more in-depth post-simulation analysis is needed to get MFPT values. However, the simulation itself is very similar.

Just like in chapter 2, we are going to run an FFPilot simulation of the SRG model. In fact, the setup of the simulation input file will be almost identical to that used in the previous chapter. The only difference comes in how the simulation output options are set. A large quantity of data is required to calculate the probability landscape of any given system. When attempting to calculate a highly accurate landscape, the size of the required simulation output files can easily climb into the terrabytes. Thus, landscape output is turned off by default.

Three different types of FFPilot simulation output are required in order to calculate a landscape. Here are the parts of each that are required for the landscape calculation (see Sec 2.2 for full explanations):

SpeciesTimeSeries (turn on with `WriteInterval` option)
`counts` (samples of the species counts) `first_passage_times` (MFPT to each tiling edge)

FFluxStageOutputRaw (turn on with `FFluxStageOutputRaw` option)
`first_trajectory_ids` (the id of the first trajectory in each phase)
`final_trajectory_ids` (the id of the final trajectory in each phase)

FFluxStageOutputSummary (on by default)
`weights` (the phase weights)

In brief, binning together the species count samples gives a biased landscape, and the other factors can be used to reweight it into the unbiased landscape.

4.2 Set up input for FFPilot landscape simulation of SRG

In order to create the needed simulation input file, first follow the procedure given in Secs 3.2.1-3.2.3, but do not follow the instructions for setting the options given in Sec 3.2.5. Instead, we will set the options needed to get all of the landscape output.

4.2.1 Enable landscape output via the FFPilot options

The following Python script sets all of the options needed for the landscape simulation:

```
import h5py

with h5py.File('data/srg.lm') as f:
    # NB: convert any numerical types to strings before setting option values

    # the trajectory state output writing interval (in terms of the
    # internal simulation time).
    f['Parameters'].attrs['writeInterval'] = str(1.0)

    # we'll need some information from the StageOutputRaw record in order
    # to calculate the landscape
    f['Parameters'].attrs['ffluxStageOutputRaw'] = 'True'

    # set an error goal, which controls the overall accuracy of
    # the simulation. Defaults to .05
    f['Parameters'].attrs['errorGoal'] = str(.10)
```

`writeInterval`
: If set, the species counts of every trajectory will be sampled and written out at the specified interval. For example, if `writeInterval` is set to `4.0`, samples are taken when the internal time of a trajectory reaches `4.0`, `8.0`, `12.0`, etc.

`ffluxStageOutputRaw`
: `bool`, defaults to `False`. When this flag is set to `True`, it turns on output of the ”raw” stage record. By default, each stage only saves a ”summary” record to the output. The raw record contains a bunch of detailed, lower-level data that is used by FFPilot during a running simulation. The raw record is needed for certain post-simulation analyses, such as the calculation of the probability landscape.

As in Sec 3.2.5, `errorGoal` is set slightly higher than the default value (`.05`) in order to produce a faster (though less accurate) simulation for demonstration purposes.

As a rule of thumb, a good value of `writeInterval` is 1 divided smallest rate constant in the model. For the SRG model we are using, the smallest rate constant is the one controlling the decay reaction, and it has a value of 1.

4.3 Running the simulation

The simulation is run in an identical fashion to the one we performed in Sec 3.3:

```
user@host:srg_landscape$ lmes -fflux -f data/srg.lm
```

The only noticeable difference should be that the simulation output file is significantly larger than it was before. This is caused by all of the system state samples that get written to the output due to the `writeinterval` option being set.

4.4 Calculating probability landscapes of SRG from FFPilot output

4.4.1 Ingesting the phase landscapes from the simulation output

We begin our analysis of the output of the SRG landscape simulation by ingesting the state samples (and other needed data) from the raw simulation output itself. Most importantly, we will bin all of the state samples that were collected during the same phase together into the phase histograms (see Sec 4.1 for details). The following Python script uses the `robertslab.SFile` package to accomplish this:

```
"""Ingest the raw data you'll need to calculate the landscape from the
simulation output SFile
"""
import re
from robertslab.sfile import SFileProto

basinRe = re.compile('Basins/(\d*)')
stageRe = re.compile('Stages/(\w*)')
phaseRe = re.compile('Phases/(\d*)')

basinEdges = {}
basinPhaseHists = {}
basinMFPTs = {}
basinPhaseWeights = {}
basinTrajectoryCounts = {}

# open the simulation output and show a progress bar
with SFileProto.open('data/srg_-_out.sfile', progress=True) as f:
    phaseHists = {}

    # iterate through each record in the output
    for rec in f.records():
        # determine the basin and stage during which the record was written
        basin = int(basinRe.search(rec.name).group(1))
        stage = stageRe.search(rec.name).group(1)

        if stage != 'Production':
```

```
            # if this isn't a record from a production stage, skip it
            continue

        if rec.dataType.find('SpeciesTimeSeries') > -1:
            # determine the phase during which the record was written
            phase = int(phaseRe.search(rec.name).group(1))

            # fetch the phaseHists for this basin
            basinPhaseHists[basin] = phaseHists = basinPhaseHists.get(basin, dict())

            # add to the count of the appropriate phaseHist
            phaseHists[phase] = phaseHist = phaseHists.get(phase, dict())

            # increment the values of the phaseHist based on the species count
            # samples in the record
            for count in (tuple(row) for
                          row in rec.msg(unpackNDArray=True)[1]['counts']):
                phaseHist[count] = phaseHist.get(count, 0) + 1

        elif rec.dataType.find('StageOutputSummary') > -1:
            # get the first passage times and the phase weights
            # from the StageOutputSummary record
            summaryMsg = rec.msg()
            basinEdges[basin] = np.array(summaryMsg.edges)
            basinMFPTs[basin] = np.array(summaryMsg.first_passage_times)
            basinPhaseWeights[basin] = np.array(summaryMsg.weights)

        elif rec.dataType.find('StageOutputRaw') > -1:
            # get the trajectory counts from the StageOutputRaw record
            rawMsg = rec.msg()
            basinTrajectoryCounts[basin] = np.array(rawMsg.final_trajectory_ids) -  \
                                           np.array(rawMsg.first_trajectory_ids) + 1
```

In brief, the script opens the output file and iterates through the records. When it finds a record containing needed data, it performs a particular action based on the record's type:

SpeciesTimeSeries
: The script retrieves the appropriate phase landscape histogram (one is created for each combination of basin and phase), unpacks the `counts` array of the record, and then adds it to that histogram.

FFluxStageOutputRaw (turn on with `FFluxStageOutputRaw` option)
: The script calculates the count of trajectories run in each phase then stores the results.

FFluxStageOutputSummary (on by default)
: The script fetches the `weights` and the `first_passage_times` and stores them.

The outcome of this script is the creation of a number of python dictionaries that contain all of the data needed for the subsequent analyses. Each of these dictionaries contains exactly two entries, one for each basin/production stage in the simulation we just ran. The nature of these entries is as follow:

`basinPhaseHists[i]`
: The phase hists from the production stage initialized at basin i.

`basinMFPTs[i]`
: The MFPT to each tiling edge of stage i.

`basinPhaseWeights[i]`
: The phase weights of stage i.

`basinTrajectoryCounts[i]`
: The total count of trajectories, whether they succeeded or failed, run during each phase of stage i

4.4.2 How histograms are represented in this analysis

The phase landscape histograms built up by the script will each contain the species count observations collected during a single particular phase. They are represented as Python dictionaries with the following format:

KEY	VALUE
...	
$(s_{k_0}, s_{k_1}, \cdots, s_{k_S})$	x_k
$(s_{k+1_0}, s_{k+1_1}, \cdots, s_{k+1_S})$	x_{k+1}
...	

Where s_{i_j} is the ith observed state (*i.e.* count) of a system's j unique chemical species, and x_i is the number of times the ith state has been observed.

For example, from our SRG landscape simulation, the contents of the histogram created from the samples collected from phase 4 of the simulation stage initialized from basin 1 may look like this:

```
((108,), 22)
((149,), 28)
((109,), 67)
((110,), 72)
((147,), 74)
...
((124,), 514)
((122,), 517)
((121,), 521)
((127,), 522)
((120,), 532)
```

4.4.3 Basic manipulation of histograms

Before moving on to the landscape calculation proper, we'll first define a few basic functions to help manipulate the histograms:

```
"""Basic histogram manipulation functions.
Assumes that the histograms are dictionaries with keys (x0, x1, ..., xn)
that represent discrete states and values m that represent the number
of times each state was observed
"""
def addHists(h0, h1, weight=1.0):
    """Add two hists together
    """
    hsum = dict(h0)
    for obs,val in h1.items():
        h0[obs] = h0.get(obs, 0) + val*weight

    return hsum

def calcOParamHist(h, oparam):
    """Produce a new hist from `h` in which all of the states have been
    transformed to their corresponding order parameter value.
    Transforms the states using the passed-in `oparam` function
    """
    oparamHist = {}

    for count,val in h.items():
        opval = oparam(count)
        oparamHist[opval] = oparamHist.get(opval, 0) + val

    return oparamHist

def normHist(h, scale=1.0):
    """Normalizes a hist such that sum(hist.values())==1
    """
    histsum = sum(list(h.values()))
    return {obs:scale*val/histsum for obs,val in h.items()}

def sparseToDense1D(h):
    """Takes a 1D hist in the dictionary (ie sparse) representation
    and produces an array (ie dense) representation of the same hist
    """
    # sort the (observation, val) pairs in the sparse histogram
    hsorted = sorted([(obs[0], val) for obs,val in h.items()])

    # split the observations and values into their own sequences
    edges,dense = [np.array(data) for data in zip(*hsorted)]

    return dense,edges
```

These 4 functions are general histogram operations:

`addHists`
: Adds the values from two histograms `h0` and `h1` together into a new histogram. For example:

```
          h0                    h1                addHists(h0,h1)

    [0, 0, 0] → [150]    [0, 0, 0] → [59]      [0, 0, 0] → [209]
    [0, 0, 1] → [141]    [0, 0, 1] → [148]     [0, 0, 1] → [289]
    [0, 1, 0] → [132]    [0, 1, 0] → [152]     [0, 1, 0] → [284]
    [0, 1, 1] → [86]     [0, 1, 1] → [139]     [0, 1, 1] → [225]
    [1, 0, 0] → [90]     [1, 0, 0] → [143]     [1, 0, 0] → [233]
    [1, 0, 1] → [136]    [1, 0, 1] → [154]     [1, 0, 1] → [290]
    [1, 1, 0] → [130]    [1, 1, 0] → [123]     [1, 1, 0] → [253]
    [1, 1, 1] → [135]    [1, 1, 1] → [82]      [1, 1, 1] → [217]
```

`calcOParamHist`
: Takes a histogram of state's (*i.e.* species counts) and converts it to a histogram of order parameter values. Uses the passed-in function `oparam` to perform the conversion.

`normHist`
: Normalizes the values of the histogram. The normalized value $\bar{x}_k$ associated with the kth observed state is found with the following simple formula:

$$\overline{x_k} = \frac{x_k}{\sum_k x_k} \tag{4.1}$$

where x_k is the kth unnormalized observation value. The values of the normalized histogram are gauranteed to sum to 1.0, meaning that a normalized histogram represents a formal probability distribution.

`sparseToDense1D`
: Given that `h` is a 1D sparse histogram (for example, any histogram returned by `calcOParamHist` when a 1D order parameter is passed in), this function will convert it to a dense histogram. This dense histogram takes the form of two 1D arrays, one for the observations and one for the values.

We will also define a Python version of the 1D order parameter $\mathbf{A}$:

```python
"""Order parameter function(s). These convert a species count to an
order parameter value.
"""
def oparamAlpha(count):
    alpha = sum(count[id]*coeff for id,coeff in zip((0,), (1,)))

    # return as a tuple
    return alpha,

oparam1D = oparamAlpha
```

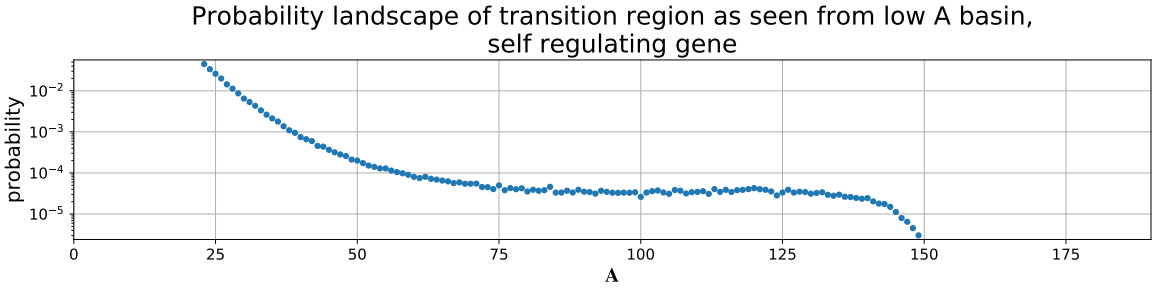
`oparamAlpha`
: Takes a species count (as a tuple) for input and returns the value of the order parameter (also as a tuple). For SRG, it may seem trivial and unnecessary to store species counts/order parameter values as tuples, since they're always single-valued, or to transform them using a function, since the species count and the order parameter always have the same value.

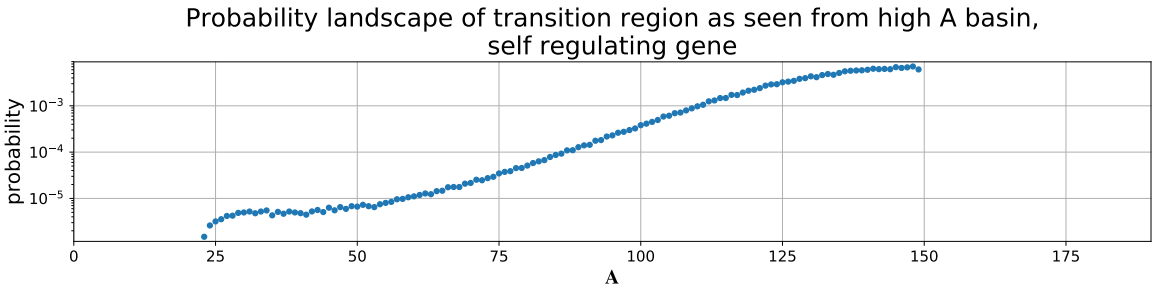
However, writing the code this way makes most of it reusable when we start dealing with systems with higher-dimensional state spaces and order parameters in the next chapter.

4.4.4 Calculate the stage histograms

The first step to calculating the landscape is to take the phase hists from all of the phases that were run during each of the stages and combine them into stage hists. The landscape weights $\mathbb{I}_i$ needed to combine the phases hists within a stage can be calculated from Eq 2.3 (see Sec 2.2.2 for details). The following Python script will calculate the $\mathbb{I}_i$ values, weight the phase hists, and then combine them into two stage hists:

```
"""create the stage hists by reweighting and combining the phase hists.
One stage hist will be created per basin set in the simulation's input
"""
basinIDs = sorted(basinPhaseHists.keys())

basinStageHists = {}
for bid in basinIDs:
    phaseHists = basinPhaseHists[bid]
    phaseWeights = basinPhaseWeights[bid]
    trajectoryCounts = basinTrajectoryCounts[bid]

    # calculate the weight the landscape sampled during each phase will have
    landscapePhaseWeights = np.ones(phaseWeights.size, dtype=float)
    landscapePhaseWeights[2:] = phaseWeights[1:-1]
    landscapePhaseWeights = landscapePhaseWeights.cumprod()
    landscapePhaseWeights[1:] /= trajectoryCounts[1:]

    # calculate the stage hist by combining all of the phase hists
    stageHist = {}
    for phaseID,weight in enumerate(landscapePhaseWeights[1:].flat):
        addHists(stageHist, phaseHists[phaseID+1], weight=weight)

    basinStageHists[bid] = stageHist
```



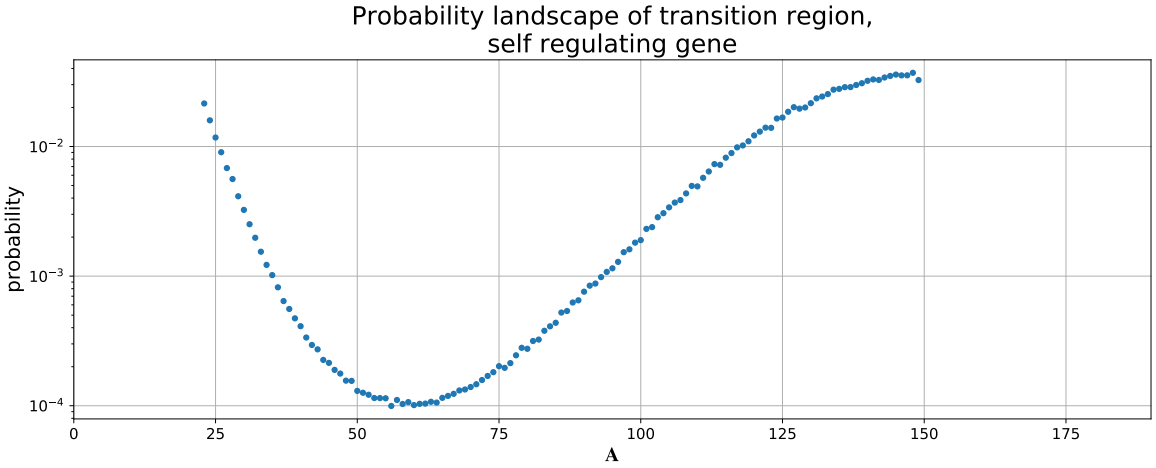


4.4.5 Combine the stage landscapes into the transition landscape

Now we find the transition region landscape by taking the sum of the two stage landscapes weighted by the stage landscape factors $\mathbb{S}_i$. The following script find the $\mathbb{S}_i$ values and performs the combination of hists:

```
"""calculate and combine the stage hists from the two basins in order
to get an unbiased hist of the transition region (ie the region
covered by the tiling used in the simulation)
"""
basinIDs = sorted(basinPhaseHists.keys())

# calculate the weight that the stage hist from each basin
# should have in the combined hist
phaseZeroWeights = {i:basinPhaseWeights[i][0] for i in basinIDs}
mfptOveralls = {i:basinMFPTs[i][-1] for i in basinIDs}

# combine the stage hists into the transition region hist
transitionHist = {}
for bid in basinIDs:
    # calculate the weight each stage hist should have in the combined hist
    stageWeight = mfptOveralls[bid]/((mfptOveralls[0]
                                      + mfptOveralls[1])*phaseZeroWeights[bid])

    # weight and add the stage hists
    addHists(transitionHist, basinStageHists[bid], weight=stageWeight)

# normalize the transition hist
transitionHist = normHist(transitionHist)
```

We now have an unbiased landscape of the SRG, though for now it only spans the transition region in between the basins:



4.4.6 Fit together the complete landscape

The final step of assembling the complete landscape is also the most complicated. It proceeds in roughly 3 steps:

1. Sort all of the phase zero samples into two histograms
One with the samples to the left of the transition minimum, and one with all of the sample to the right. The following script will accomplish this:

```
# find an interval (in terms of the 1D order parameter) containing
# the transition state minimum
width = 20
transitionMinOpval = sorted(transitionHist.items(),
                            key=lambda x: x[1])[0][0][0]

transitionMinLeft = transitionMinOpval - width/2
transitionMinRight = transitionMinOpval + width/2

# Sort the phase zero data into a left and right hist (with respect to
# the transition region minimum)
phaseZeroHists = [basinPhaseHists[i][0] for i in basinIDs]
phaseZeroLeftHist = {}
for count,val in phaseZeroHists[0].items():
    if oparam1D(count)[0] < transitionMinLeft:
        phaseZeroLeftHist[count] = val
phaseZeroRightHist = {}
for count,val in phaseZeroHists[1].items():
    if oparam1D(count)[0] >= transitionMinRight:
        phaseZeroRightHist[count] = val
phaseZeroSideHists = [normHist(h) for h in (phaseZeroLeftHist, phaseZeroRightHist)]
```

2. Fit the left and right phase zero histograms onto the transition region landscape
Independently, we find the weights that will smoothly fit both the left and right phase zero histograms

to the transition landscape. We do this using a straightforward minimization approach, with a least-squares minimizer and a Jenson-Shannon divergence*. The following script will perform this fitting and report the weights:

```
import scipy.optimize as opt

# fit the left and right hists seperately to the transition hist
transitionHist1D = calcOParamHist(transitionHist, oparam1D)
sideLandscapeWeights = []
for pzHist in phaseZeroSideHists:
    pzHist1D = calcOParamHist(pzHist, oparam1D)

    # define a Jensen-Shanon divergence (symmetrized KL divergence) function
    # for the minimizer
    def _jsdiv(x):
        div = 0

        for opval in (opval for opval in transitionHist1D.keys()
                      if opval in pzHist1D):
            transval,pzval = transitionHist1D[opval], x*pzHist1D[opval]

            # iterate only over states in which both hists contain
            # at least some density
            if transval==0 or pzval==0:
                continue

            meanval = .5*(transval + pzval)
            div += .5*(transval*np.log(transval/meanval)
                       + pzval*np.log(pzval/meanval))

        return div

    # start the fitting weight at 2.0
    weightFit0 = 2.0

    # fit by minimizing the difference between the transition hist and the section
    # of the phase zero hist that overlaps with the transition hist
    weightFitOut = opt.minimize(_jsdiv, x0=weightFit0, method='Nelder-Mead')
    sideLandscapeWeights.append(weightFitOut.x[0])

# print some info about the fit of the left and right sides of the landscape
print('The left side of phase zero will have a weight of: %.3f'
      % sideLandscapeWeights[0])
print('The right side of phase zero will have a weight of: %.3f'
      % sideLandscapeWeights[1])
```

3. Patch together the phase zero histograms and the transition landscape

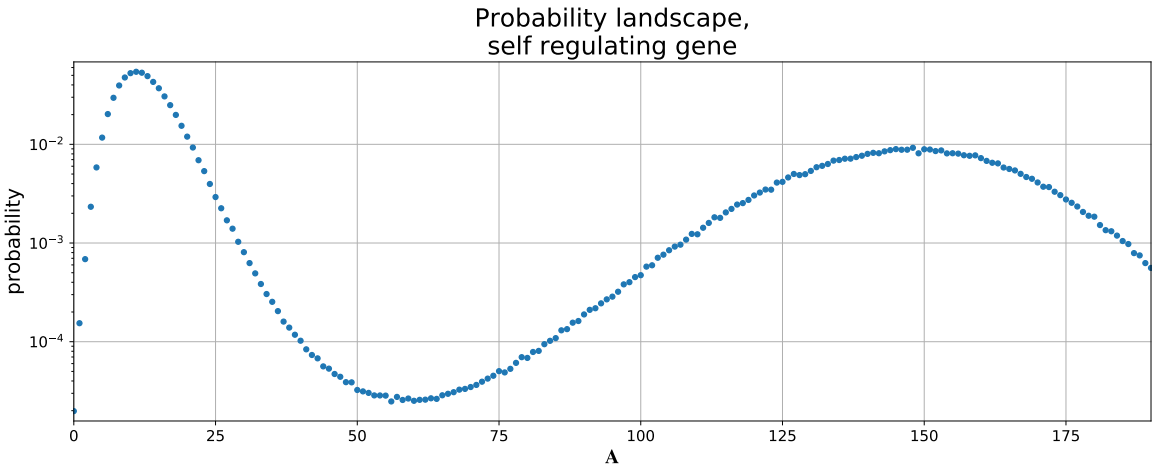
Finally, we patch the weighted phase zero histograms together with the transition landscape in order to produce the complete landscape. The following script performs this patching:

*a symmetrized form of the Kullback-Leibler divergence

```
# initialize the hist that will cover the entire landscape by
# starting with the transition region hist
landscapeHist = dict(transitionHist)

# get the range of the transition region, in terms of the
# 1D order parameter values
oparamvals = [oparam1D(count) for count in landscapeHist.keys()]
opmin,opmax = min(oparamvals), max(oparamvals)

# add the non-overlapping data from the left and
# right hists to the transition hist
for count,val in phaseZeroSideHists[0].items():
    opval = oparam1D(count)
    if opval < opmin:
        if count in landscapeHist: raise KeyError
        landscapeHist[count] = sideLandscapeWeights[0]*val
for count,val in phaseZeroSideHists[1].items():
    opval = oparam1D(count)
    if opval > opmax:
        if count in landscapeHist: raise KeyError
        landscapeHist[count] = sideLandscapeWeights[1]*val
landscapeHist = normHist(landscapeHist)
```

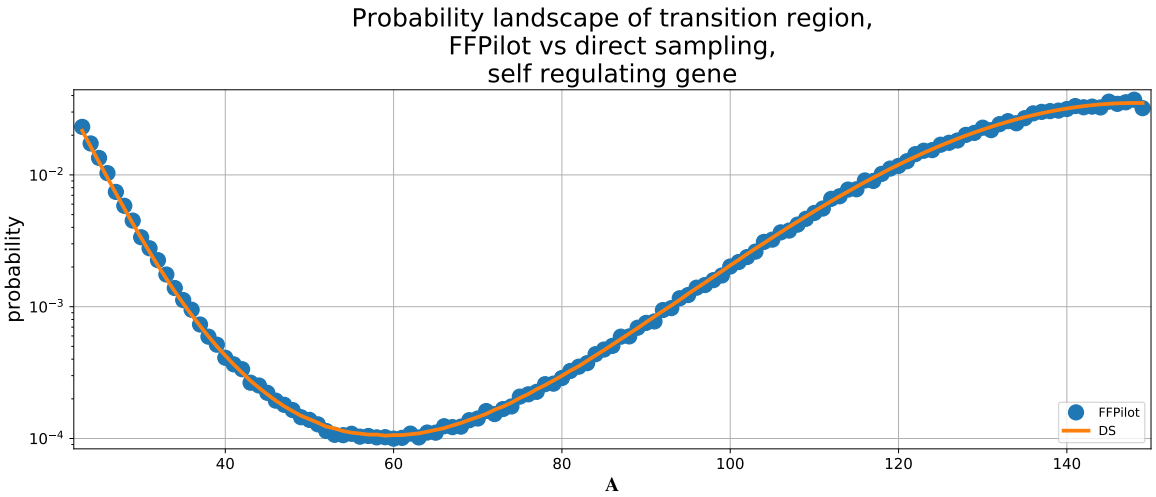
If any bin in the phase zero histograms overlaps with any bin in the transition landscape, we discard the information from the phase zero histograms in favor of that in the transition landscape, on the assumption that the transition landscape will in general be better sampled.

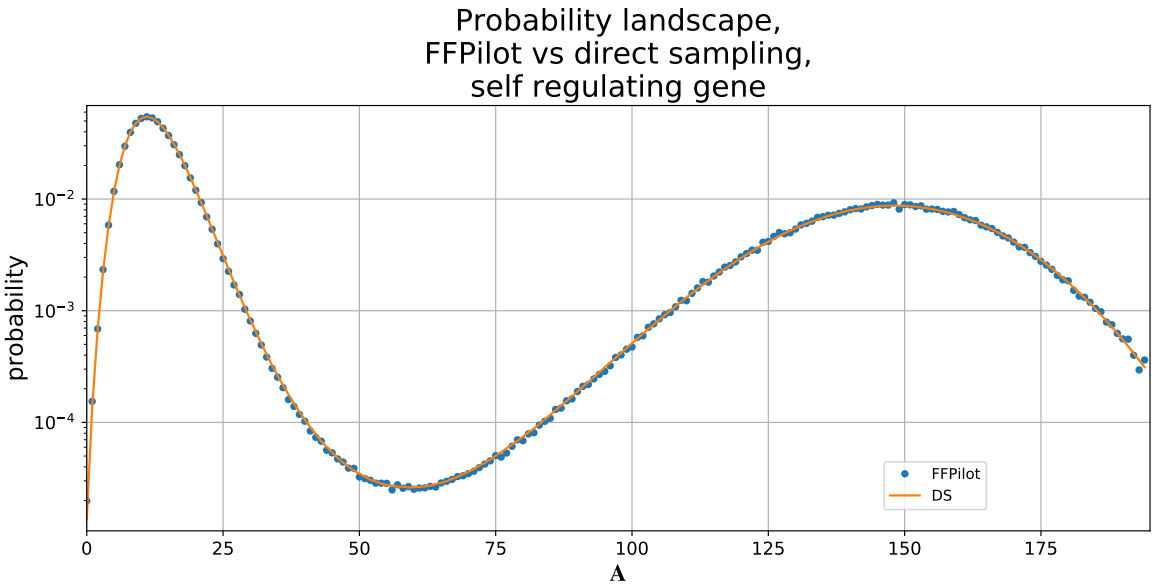
Now that it's all put together, the complete unbiased landscape looks like this:



4.4.7 Comparison of landscapes produced using FFPilot and direct sampling

Shown below are a comparison of landscapes calculated using a (blue dots) 10% error goal FFPilot simulation and (orange line) direct sampling simulation that collected 10^9 state samples.





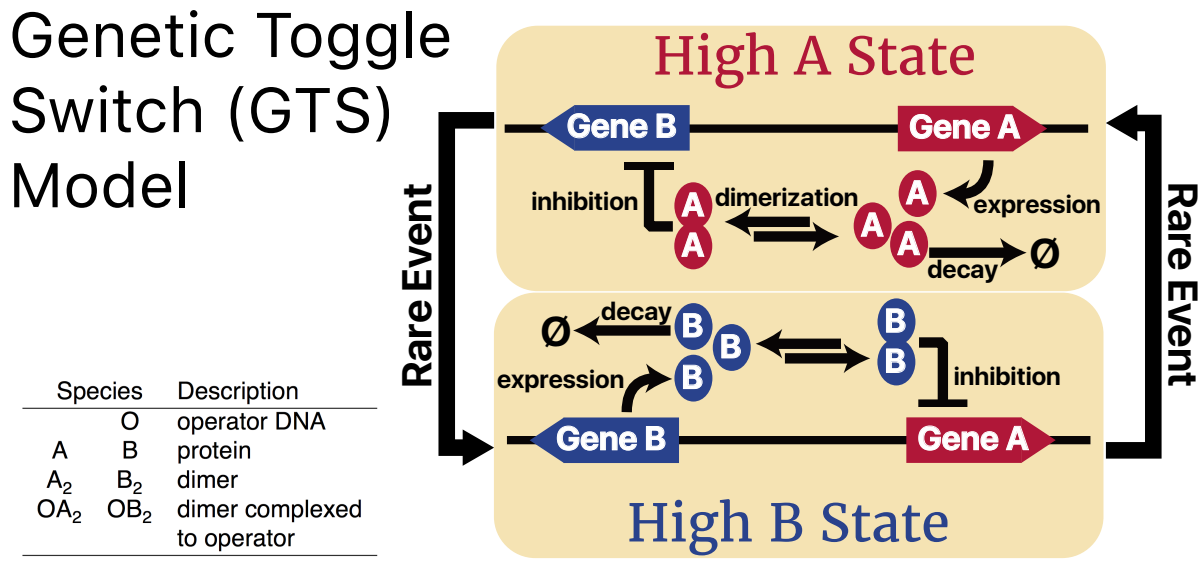
Although the FFPilot landscape is in this case slightly noisier, overall there is a high level of agreement between the FFPilot and direct sampling landscapes. This is made more impressive by the fact that, in terms of real wall-clock time, the FFPilot simulation ran to completion about 140x faster than the direct sampling one (~ 30 seconds vs ~ 70 minutes).

Chapter 5

Using FFPilot to simulate systems with complex, high-dimensional state spaces: genetic toggle switch

5.1 Overview

In this chapter, we'll be working with the genetic toggle switch (GTS). genetic toggle switch (GTS) is to the study of genetic regulatory systems what the hydrogen atom was to the study of quantum mechanics. It is one of the simplest truly biphasic systems that has actually been experimentally instantiated in living cells [3]. GTS models come in many different variants, and we will work with one called the exclusive genetic toggle swtich. It consists of 7 distinct chemical species that participate in 14 separate reactions.



Species		Description
	O	operator DNA
A	B	protein
A_2	B_2	dimer
OA_2	OB_2	dimer complexed to operator

Reactions		Rate Constants		Description
Protein A	*Protein B*	*forward*	*reverse*	
$O \longrightarrow O + A$	$O \longrightarrow O + B$	θ		expression
$A \longrightarrow \varnothing$	$B \longrightarrow \varnothing$	$\theta/4$		decay
$2\,A \rightleftharpoons A_2$	$2\,B \rightleftharpoons B_2$	5	5	dimerization
$O + A_2 \rightleftharpoons OA_2$	$O + B_2 \rightleftharpoons OB_2$	5	1	operator binding
$OA_2 \longrightarrow OA_2 + A$	$OB_2 \longrightarrow OB_2 + B$	θ		bound expression

Protein A and protein B are both produced from a single piece of operator DNA. A and B can both dimerize, and each dimer can bind back to the operator DNA. Only one dimer can bind to the DNA at a time. When the DNA is unbound, it will produce both A and B at an equal rate. However, when a dimer of one of the proteins is bound, the DNA will only produce more of that same protein. This means that GTS spends the vast majority of its time in either a high A, low B state (state $\mathcal{A}$), or a high B, low A state (state $\mathcal{B}$).

SRG has 1 unique chemical species, and thus a 1D state space. GTS has 7 unique species and a 7D state space, and so is inherently a much more complex system to simulate and analyze. In particular, complex sources of error emerge in higher dimensional systems that aren't present in 1D systems like SRG. Helpfully, GTS has a feature that makes it easy to examine simulation error: it's perfectly symmetric in terms of proteins A and B. In fact, because the species and reactions containing A are identical to those containing B (aside from, of course, the change in protein), the MFPT of the $\mathcal{A} \to \mathcal{B}$ transition is equal to that of $\mathcal{B} \to \mathcal{A}$, and the probability landscape of GTS is perfectly symmetrical. The deviation from this 2-fold symmetry can thus be used as a simple measure of simulation accuracy.

Over the course of the previous chapters in this tutorial we established the theory and protocols for running and analyzing FFPilot simulations. In this chapter we'll focus on error, and how to think about simulation accuracy in general.

5.2 FFPilot simulation input for GTS

The protocol for setting up the input for an FFPilot simulation of GTS is very similar that used in the earlier chapters for setting up simulations of SRG. Essentially, certain array values, such as `SpeciesCoefficients` and `Basins`, which were 1D for SRG are 7D for GTS. Although this may sound complex, in practice it is not much more complex than setting up SRG.

Once again, we're going to use the `lm_sbml_import` to convert a `.sbml` file to an `.lm` file, and then use the `h5py` Python package to add the needed order parameter and tiling.

First, we run the `lm_sbml_import` utility on `genetic_toggle_switch.sbml` in order to produce `gts.lm`. Open a terminal, then `cd` to `notebooks/gts` and then execute:

```
user@host:gts$ mkdir -p data
user@host:gts$ lm_sbml_import data/gts.lm ../genetic_toggle_switch.sbml
```

Next we set the order parameter, the tiling, and the FFPilot simulation options needed for landscape output using the following Python scripts:

order parameter

```
import h5py

with h5py.File('data/gts.lm') as f:
    # remove any existing OrderParameters group
    if 'OrderParameters' in f.keys():
```

```
        del f['OrderParameters']

    # create the OrderParameters group
    oparams = f.create_group('OrderParameters')

    # add the subgroup for order parameter 0
    oparam0 = oparams.create_group('0')

    # set this order parameter's ID to 0, and its Type to 0
    oparam0.attrs['ID'] = 0
    oparam0.attrs['Type'] = 0

    # add the SpeciesIDs. This array should always be of type `int`
    speciesIDs = np.array([0, 1, 2, 3, 4, 5], dtype=int)
    oparam0.create_dataset('SpeciesIDs', data=speciesIDs)

    # add the SpeciesCoefficients. This array should always be of type `float`
    speciesCoefficients = np.array([-1, -2, -2, 1, 2, 2], dtype=float)
    oparam0.create_dataset('SpeciesCoefficients', data=speciesCoefficients)
```

tiling

```
import h5py

with h5py.File('data/gts.lm') as f:
    # remove any existing Tilings group
    if 'Tilings' in f.keys():
        del f['Tilings']

    # create the Tilings group
    tilings = f.create_group('Tilings')

    # add the subgroup for tiling 0
    tiling0 = tilings.create_group('0')

    # set this tiling's ID to 0, its OrderParameterID to 0, and its Type to 0
    tiling0.attrs['ID'] = 0
    tiling0.attrs['OrderParameterID'] = 0
    tiling0.attrs['Type'] = 0

    # add the Basins. There are 7 chemical species in this model,
    # and we are going to set 2 initial basins, so Basins will be a 2 x 7 array
    basins = np.zeros((2,7), dtype=int)
    basins[0,:] = [4, 16, 1, 0, 0, 0, 0]
    basins[1,:] = [0, 0, 0, 4, 16, 1, 0]
    tiling0.create_dataset('Basins', data=basins)

    # add the Edges
    edges = np.linspace(-27.0, 27.0, num=13)
    tiling0.create_dataset('Edges', data=edges)
```

FFPilot simulation options

```
import h5py

with h5py.File('data/gts.lm') as f:
    # NB: convert any numerical types to strings before setting option values

    # set a higher than default error goal (.05) for a quicker simulation
    f['Parameters'].attrs['errorGoal'] = str(.10)

    # turn on output of sage raw records
    f['Parameters'].attrs['ffluxStageOutputRaw'] = str(True)

    # 1 over the decay rate (.25) of the genetic toggle switch
    f['Parameters'].attrs['writeInterval'] = str(4.0)
```

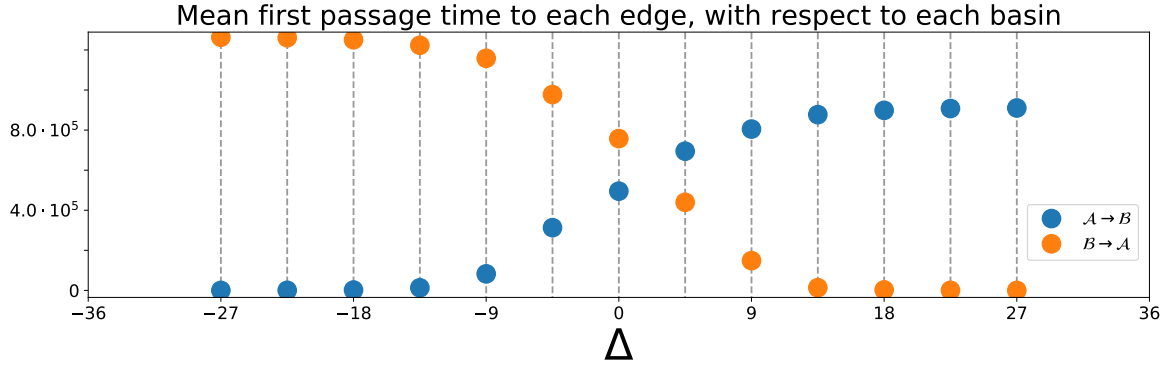
5.3 Running the simulation

No surprises here, as the GTS simulation can be executed with the same basic command that was used to run the SRG simulations:

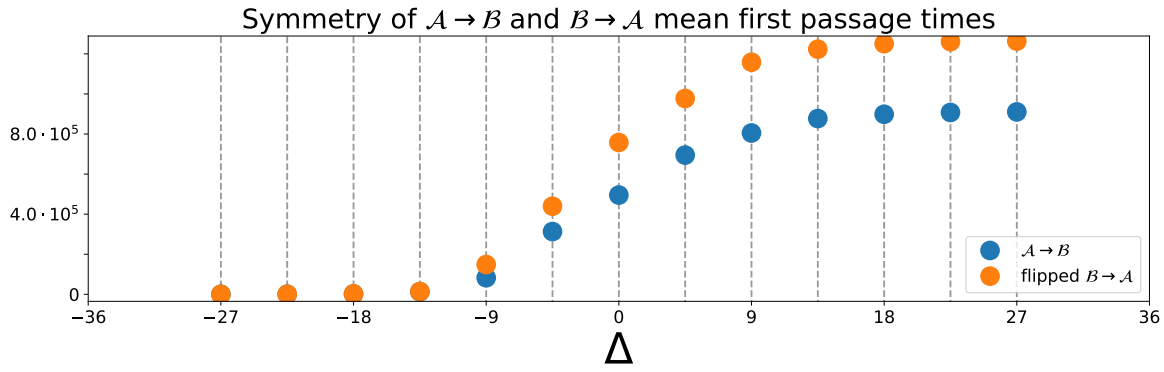
```
user@host:gts$ lmes -fflux -f data/gts.lm
```

5.4 MFPT to each edge of GTS tiling

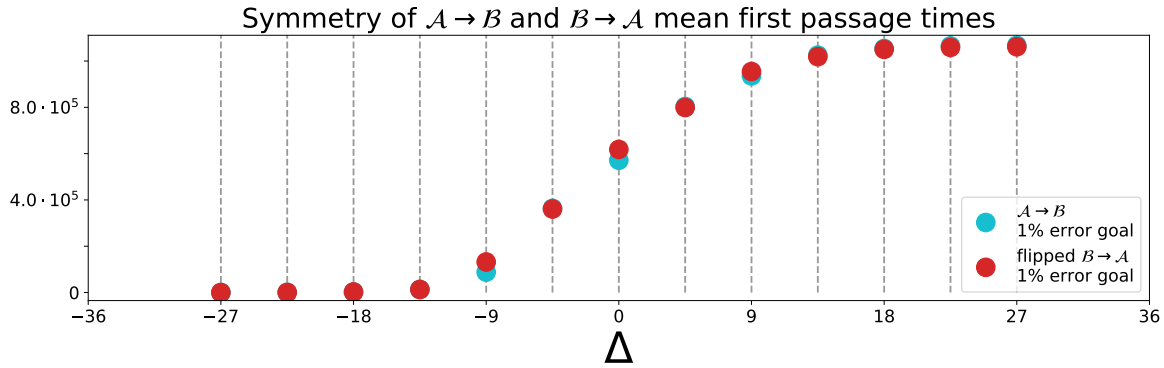
As with our SRG simulations, the MFPT values of GTS can be fetched directly from the simulation stage summary records. Follow the instructions given in Sec 3.6.1 in order to retrieve the MFPT results from the GTS simulation we just ran. Plotting those values should yield something along the lines of:



There are two sets of GTS MFPT values, those collected when switching $\mathcal{A} \to \mathcal{B}$ and when switching $\mathcal{B} \to \mathcal{A}$. Regardless of what kind of simulation was used to estimate them, they should be perfectly symmetrical. This means that if one set is flipped with respect to its edges, the two sets would then be equal. However, complete equality is unlikely to be observed in the results from a 10% error goal simulation. Instead, it is far more likely that some deviation between the two sets of MFPTs will occur:



When GTS is simulated using a 1% error goal, much less deviation occurs between the results from the $\mathcal{A} \to \mathcal{B}$ and $\mathcal{B} \to \mathcal{A}$ processes:



As can be seen above, even though some deviations do begin to develop in phases immediately before or around the transition midpoint (*i.e.* $\Delta = 0$), the simulation recovers from those deviations, and they do not persist into the overall basin-to-basin MFPT estimates. This can be though of as a signature of the technique which FFPilot uses to optimize computational efficiency. The optimizing equation that drives FFPilot tends to bias simulations against running too many trajectories during the most expensive phases (for GTS, the most expensive phases are indeed those in the vicinity of the transition midpoint), and makes up for any errors this might introduce by causing more simulations to be run during the least expensive phases.

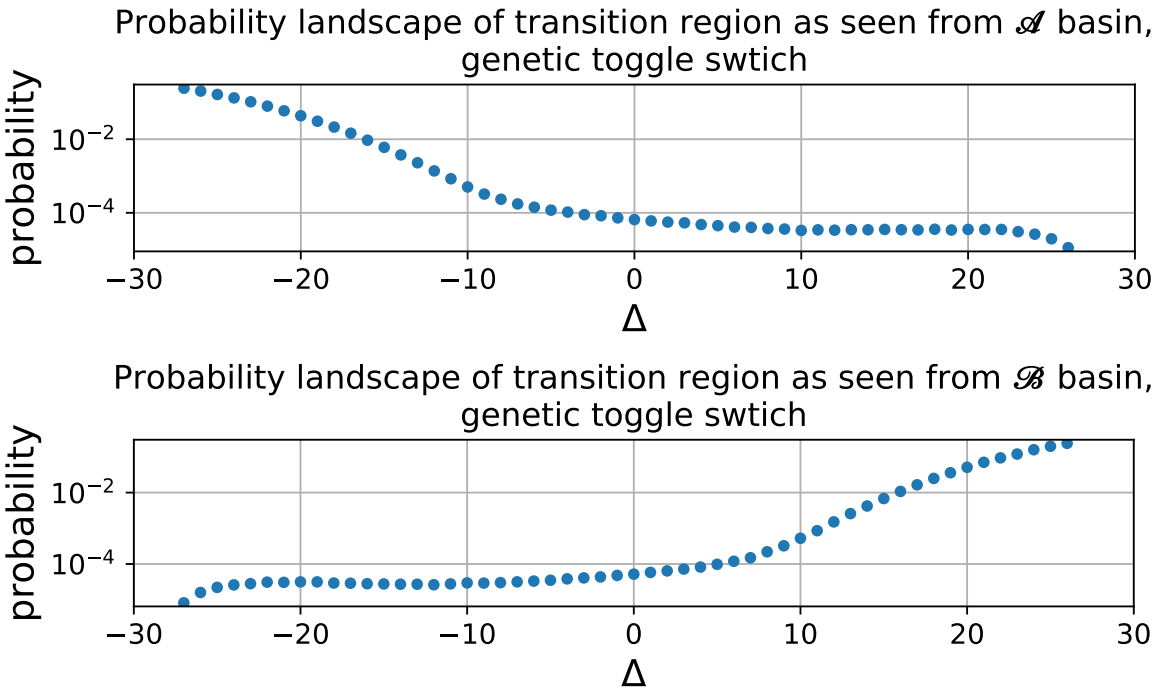
5.5 Landscape of the transition region

The code we introduced last chapter in Sec 4.4 for calculating the landscape of SRG is flexible enough to be reused almost in its entirety when calculating the landscape of the very different GTS. The only major differences are that we will use the GTS specific 1D order parameter **Δ**, and also define a 2D order parameter {**A**, **B**} to do some fancy plotting with:

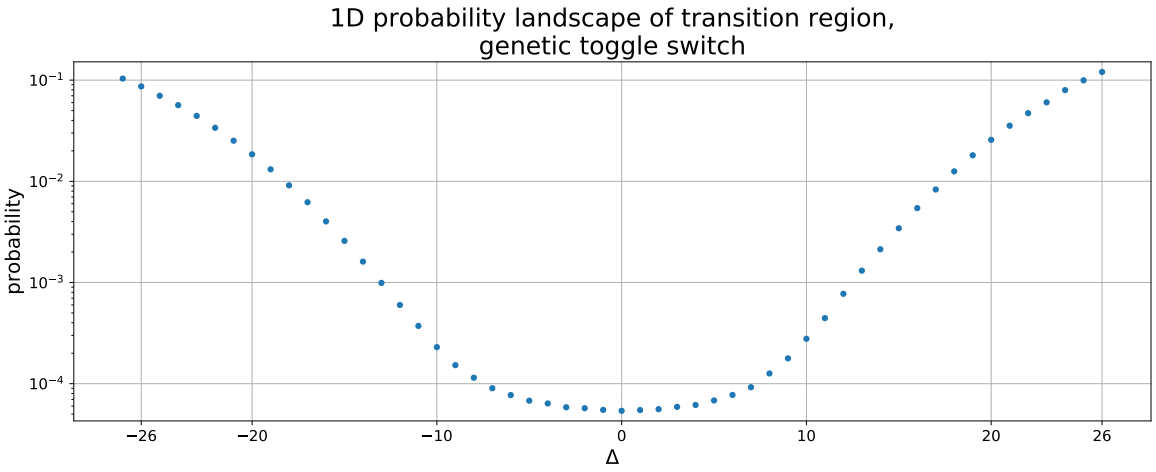
```
"""Order parameter functions. These convert a species count to an
order parameter value.
"""
def oparamDelta(count):
    delta = sum(count[id]*coeff for id,coeff in zip((0,1,2,3,4,5), (-1,-2,-2,1,2,2)))

    # return as a tuple
    return delta,

def oparamAB(count):
    A = sum(count[id]*coeff for id,coeff in zip((0,1,2), (1,2,2)))
    B = sum(count[id]*coeff for id,coeff in zip((3,4,5), (1,2,2)))

    return A,B

oparam1D = oparamDelta
oparam2D = oparamAB
```

5.5.1 Calculating the transition region landscape

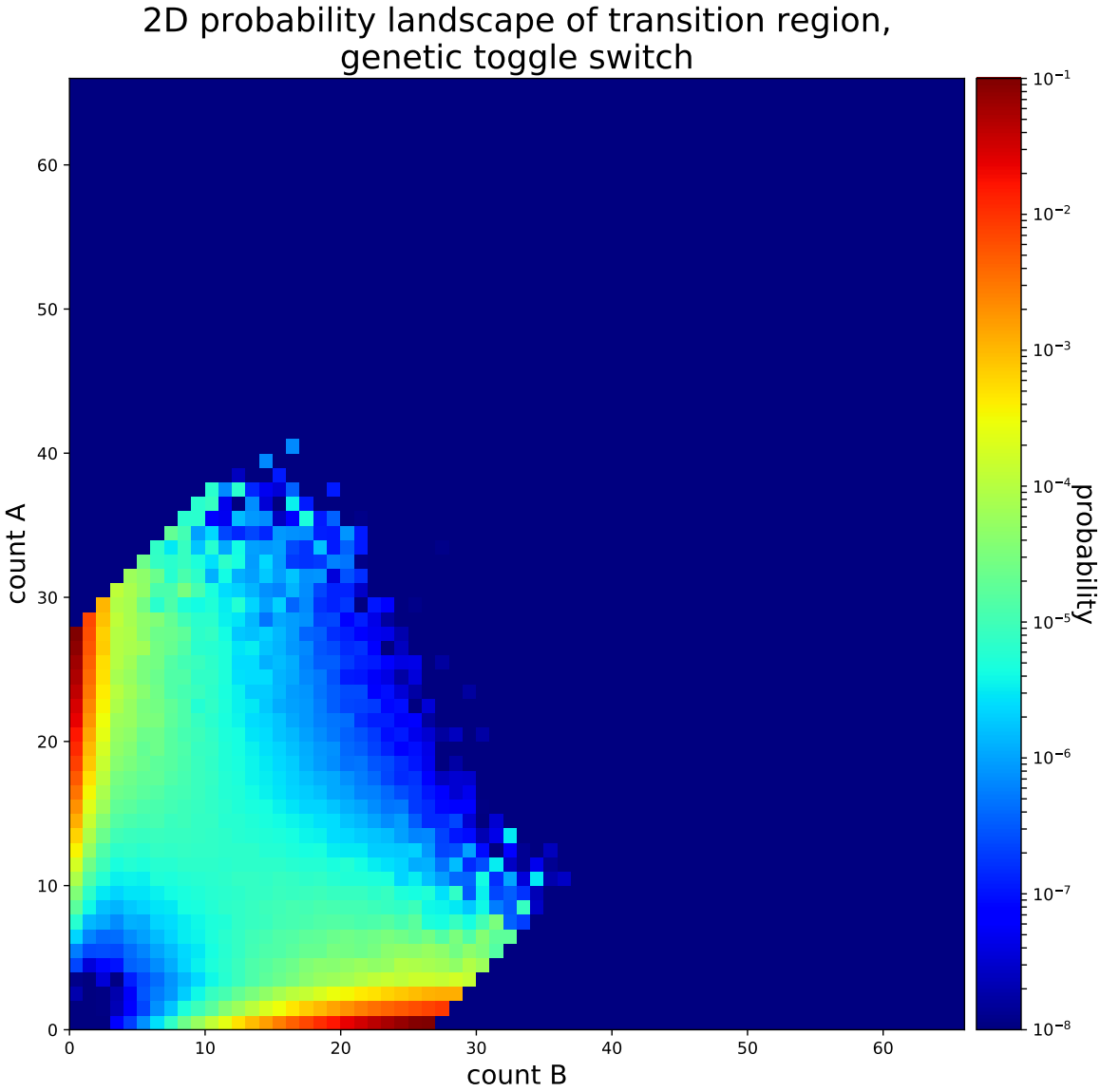
The code from Sec 4.4.4 can be used to produce the GTS stage histograms:



The GTS stage histograms can then be combined into a single transition region landscape using the code from Sec 4.4.5:



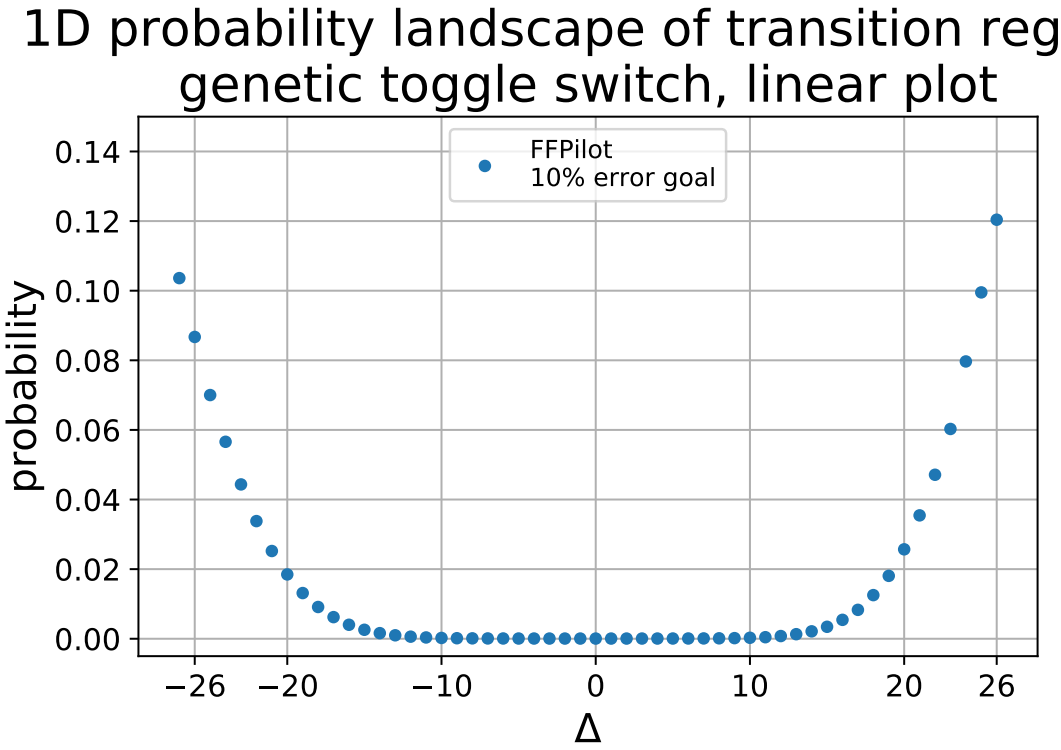
A great deal more detail can be gleaned when the transition is plotted using the 2D {**A**, **B**} order parameter:



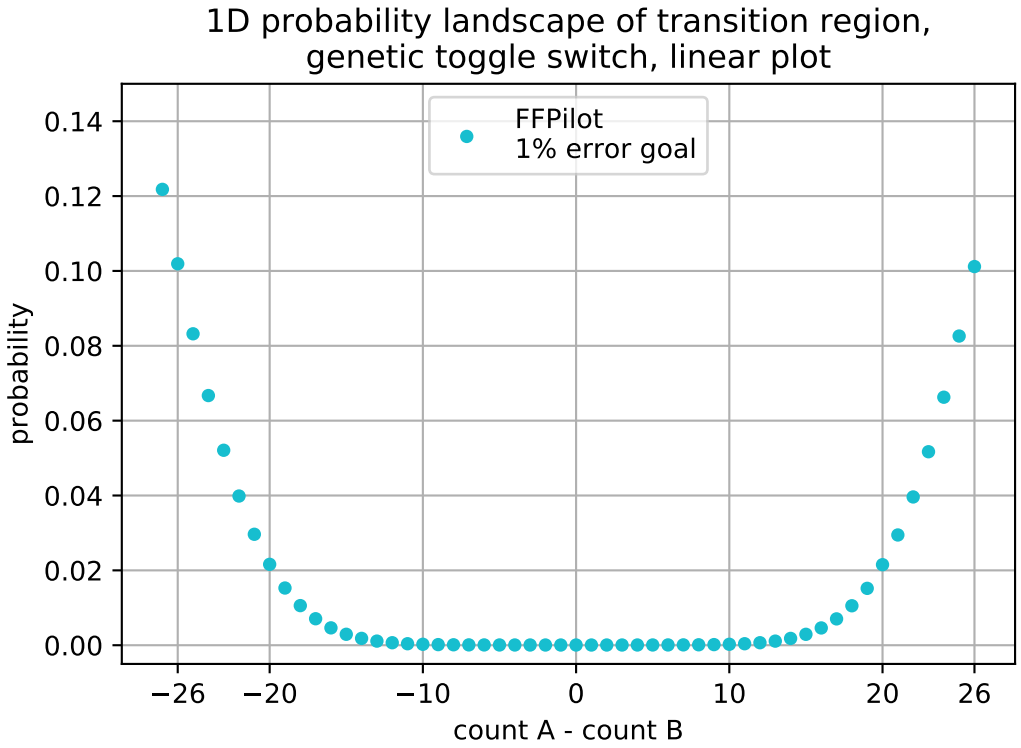
5.5.2 Accuracy of the calculated transition region landscape

In terms of the 1D order parameter Δ (ie count A − count B), the GTS landscape is symmetric over the origin. In terms of the 2D order parameter {count A, count B}, GTS is symmetric over the line $x = y$. These symmetries are important, since they offer a simple way to test the accuracy of our landscape assembly procedure (which includes both the simulation and the analysis). For any landscape we calculate, the closer its two halves are to perfect symmetry the more accurate the assembly procedure was.

Symmetry, or lack thereof, is usually easier to spot in plots of 1D order parameters. Though somewhat analytically crude, a comparison of any two symmetric points $-x, x$ in the landscape can be used as a simple, quantitative measure of the landscape's overall accuracy. A sense of how FFPilot error goal affects the accuracy of the landscape can be gained by comparing the two symmetric points $\Delta = -26$ and $\Delta = 26$. When a landscape is calculated from a 10% error goal FFPilot simulation, the divergence between the probabilities of -26 and 26 tends to be quite large, on the order of $10 - 20\%$:



On the other hand, as in the plot shown below, the probabilities of -26 and 26 are nearly equal when calculated using data from a 1% error goal simulation:

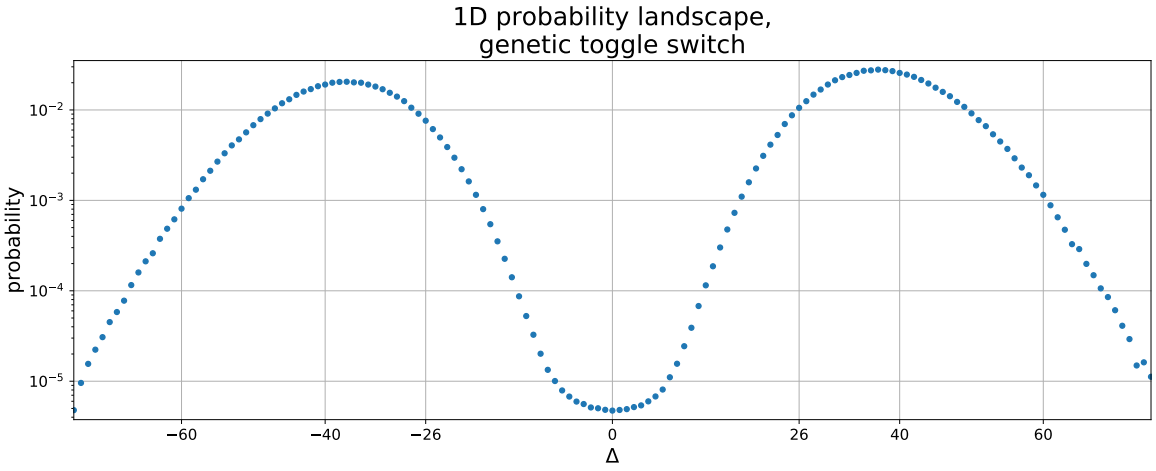


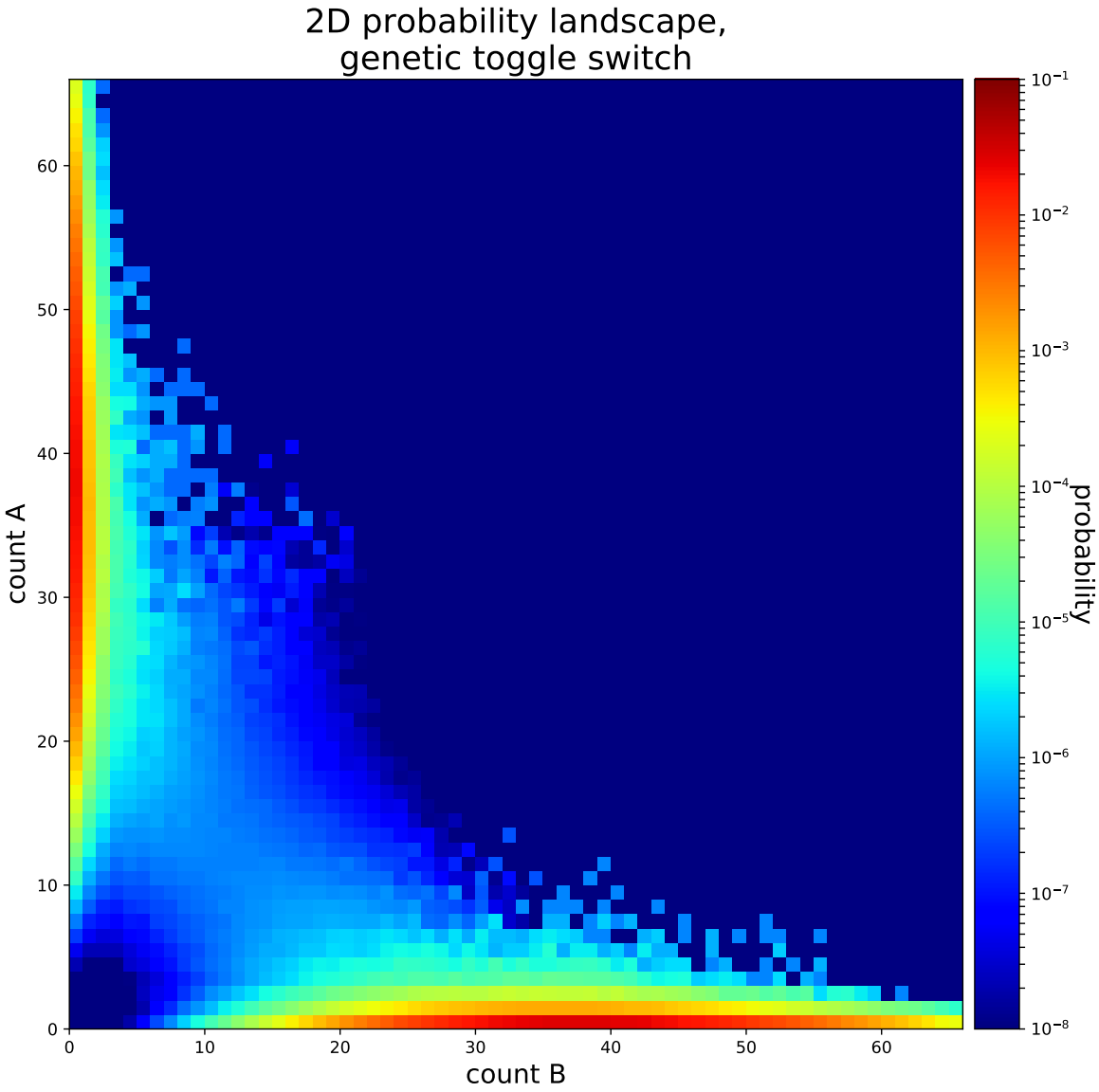
The landscape shown above was produced from the example data included with the notebook (in the `example_arrays/landscape_ffpilot_basin_%d.npz` files).

5.6 The complete unbiased landscape of GTS

5.6.1 Calculating the complete landscape

Now that we have the transition landscape assembled, the complete landscape can be calculated by simply reusing the code from Sec 4.4.6. Plotted in 1D (along the **Δ** order parameter) and 2D (along the {**A**, **B**} order parameter) it looks like:

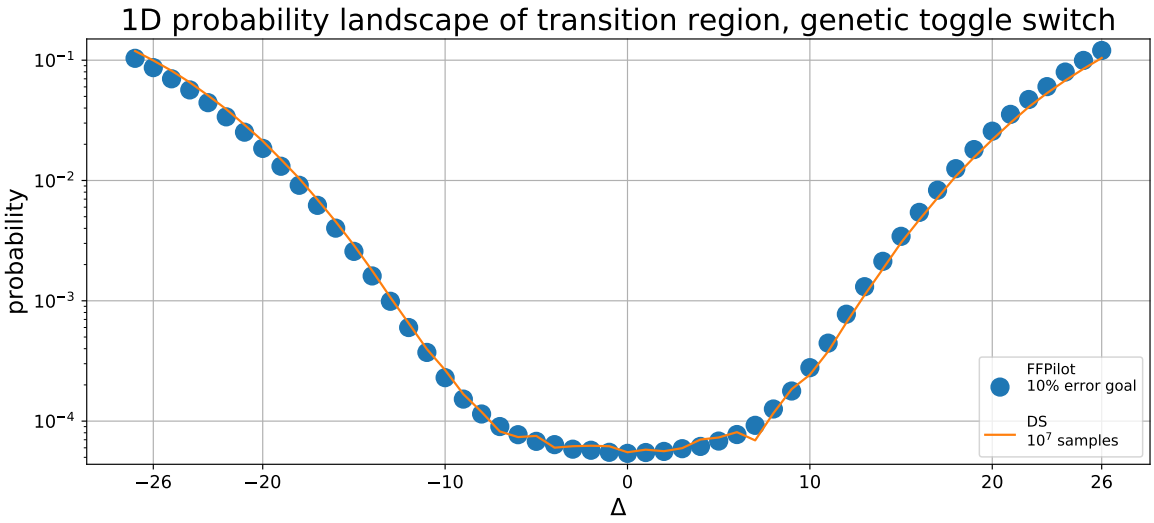




As you can see above, the fitting procedure in the code we wrote last chapter is robust and flexible enough that it is able to deal with both the SRG and the (very much different) GTS models.

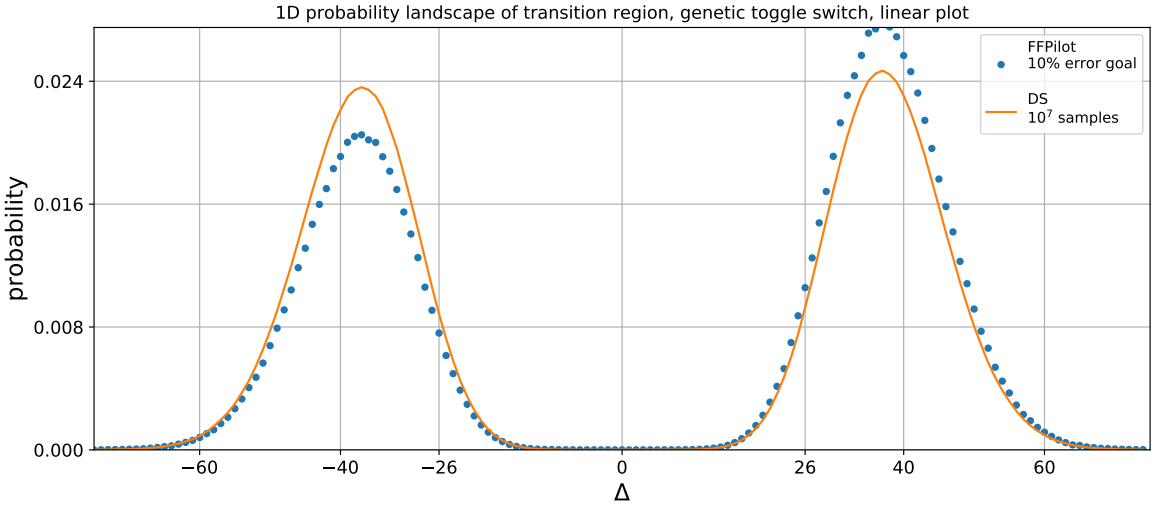
5.6.2 Accuracy of calculated complete landscape: comparison with replicate sampling

In addition to comparing them against each other, we can also gauge the accuracy of the two halves of a GTS FFPilot simulation by comparing them to the results from a direct sampling (DS) simulation. Though DS simulation is computationally inefficient, it is highly accurate. It has been shown [4] that DS simulation will converge roughly monotonically to the correct answer under a large variety of conditions. This convergence behavior, along with a history of decades of use and verification, allows DS results to be used as a sort of gold standard when examining novel simulation methods. Here is a comparison of the transition region calculated by FFPilot and DS simulations:



The landscape produced by FFPilot is much less noisy than the landscape from direct sampling in the immediate neighborhood of the transition point. In terms of landscapes, FFPilot simulation is at its most useful in any region of extremely low probability. Direct sampling tends to struggle with these low probability regions, and can only collect enough samples to map their landscapes with reasonable accuracy when vast computational resources are used.

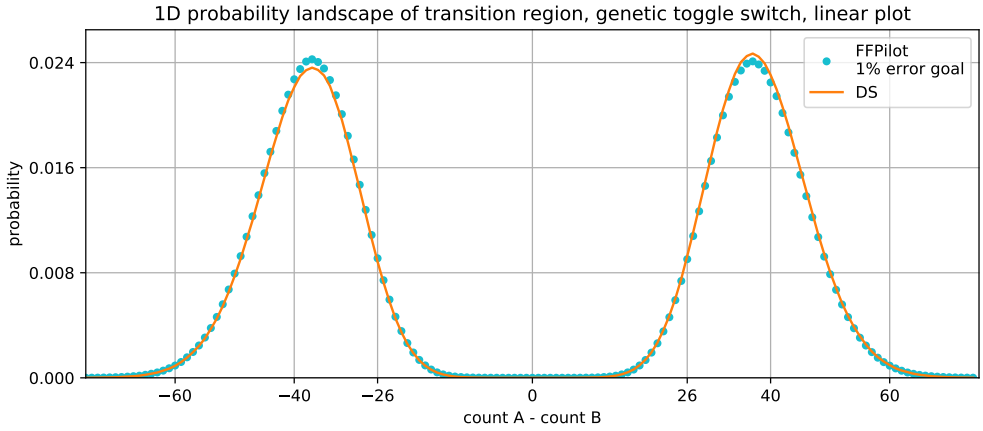
This version of the genetic toggle switch is completely symmetric in terms of A and B. Thus, the two peaks in the probability distribution should be the same height. The peak on the left corresponds to state $\mathcal{A}$ (in which there is a high count of protein A and a low count of protein B) and the peek on the right corresponds to state $\mathcal{B}$ (in which there is a high count of protein B and a low count of protein A). In terms of the 1D $\mathbf{\Delta}$, the two peeks $\mathcal{A}$ and $\mathcal{B}$ fall at -37 and 37, respectively.



In the direct sampling data we show above, the height of the state $\mathcal{A}$ and state $\mathcal{B}$ peeks differ by about $\sim 4.5\%$ relative to one another. Whether FFPilot under- or out-performs this figure depends on the error goal with which a simulation is run. Error goal is expressed in terms of a desired level of error in the estimated MFPT from a simulation, but a lower error goal also tends to reduce error in other measures as well, including this landscape symmetry measure.

In landscapes produced from FFPilot simulations run at an error goal of 10%, the two peeks can

differ by 10% or more. However, the difference in the peeks decreases dramatically for FFPilot simulations run with a 1% error goal. The landscape shown below was produced from the example data included with the notebook (in the `data_example/landscape_ffpilot_basin_%d.npz` files), which was obtained from an FFPilot simulation run with a 1% error goal:



In the landscape produced by FFPilot the heights of the $\mathcal{A}$ and $\mathcal{B}$ peeks are closely matched, and differ by less than 0.7%. In this case FFPilot outperformed direct sampling in terms of correctly reproducing the underlying symmetry of the genetic toggle switch's landscape.

Appendix A

Setting up FFPilot simulation input using the HDFView gui

A.1 Overview

A.2 Self regulating gene setup

1. Convert the `self_regulating_gene.sbml` file to the standard `.lm` simulation input format, following the instructions given in .
2. Use `HDFView` to manually add the FFPilot-specific simulation input to the resulting `self_regulating_gene.lm` file:
   (a) Open `self_regulating_gene.lm` in `HDFView`. `HDFView` is the standard file browser for `.hdf5` files and is available on the https://support.hdfgroup.org/products/java/hdfview/.
   (b) Add the order parameter:
      i. Right click on `self_regulating_gene.lm` in the column on the left, and then select `New → Group`. Enter `OrderParameters` as the new group's name and then hit `Ok`.
      ii. Right click on the new `OrderParameters` group and then repeat 2(b)i in order to create a new subgroup named `0000000`. NB: make sure the subgroup name is exactly seven zeros (internally, `LMES` uses a seven digit fixed-width format (c-style format `%07d`) for this subgroup name). .
      iii. Right click on `0000000`, choose `Show Properties`, and then click on the `Attributes` tab in the properties window. Add two attributes, `ID` and `Type`. Both should be 64-bit integer scalars, and both should have a value of 0.
      iv. Add a dataset called `SpeciesCoefficients` to the `0000000` group. It should be of type 64-bit float, and it should have a size of `1`. Double click on `SpeciesCoefficients` to open it, and then set its single value to `1.0`.
      v. Add another dataset to `0000000`. Name this one `SpeciesIDs`, set its type to 32-bit unsigned integer, and set its size to `1`. Open `SpeciesIDs` and make sure that its single value is set to 0 (which it should be by default).
   (c) Add the tiling (also called the interfaces, or the bins):
      i. Add a `Tilings` group with a `0000000` subgroup, just as you did for `OrderParameters`.

ii. Set three attributes on the `0000000` subgroup, `ID`, `OrderParameterID`, and `type`. All of them should be 64-bit integer scalars with a value of 0.

iii. Add a dataset called `Basins` to `0000000`. It should be of size 1, of type 32-bit unsigned integer, and its single value should be set to 10.

iv. Add a dataset called `Edges` to `0000000`. `Edges` should be of type 64-bit float, and should have a size of 13. The first value should be `23.0`, the last value should be `150.0`, and the remaining values should be evenly spaced in between. The easiest way to enter these values is to import them from `files/self_regulating_gene_mfpt/edges.txt`, a plain text file that contains the needed values, one per line. To import the edges open `Edges`, select `Import Data from Text File` from the `Table` menu in the upper left hand corner of `Edges`, select the `edges.txt` file, and then click okay on any prompts that pop up.

(d) No other data is required to run an FFPilot simulation. There are, however, a number of options that can be used to tweak FFPilot's execution. These options can be set by adding the appropriate attributes to the top level `Parameters` group in the input `.lm` file:

i. The overall accuracy of the simulation is controlled by the `errorGoal` option. Add an attribute to `Parameters` called `errorGoal` with a scalar float value of 0.05.

ii. By default, the output of an FFPilot simulation will consist of a single record, of type `FFPilotStageOutput`, that contains the primary results from the FFPilot production stage. For the purposes of this example simulation, we'll turn on the output of the `FFPilotStageOutputRaw` record, which contains various intermediate data. Add a `Parameters` attribute called `ffpilotStageOutputRaw` with a string value of "True" .

iii. We'll also turn on the output of the `FFPilotStageOutput` and `FFPilotStageOutputRaw` for the pilot stage. Add a `Parameters` attribute called `ffpilotPilotOutput` with a string value of "True".

List of Abbreviations

DS – direct sampling
GTS – genetic toggle switch
LMES – Lattice Microbes ES
MFPT – mean first passage time
SRG – self-regulating gene

Bibliography

[1] Valeriani, C, Allen, RJ, Morelli, MJ, Frenkel, D, Rein ten Wolde, P (2007) Computing stationary distributions in equilibrium and nonequilibrium systems with forward flux sampling. *J Chem Phys* 127:114109.

[2] Dickson, A, Warmflash, A, Dinner, AR (2009) Nonequilibrium umbrella sampling in spaces of many order parameters. *J Chem Phys* 130:074104.

[3] Gardner, TS, Cantor, CR, Collins, JJ (2000) Construction of a genetic toggle switch in Escherichia coli. *Nature* 403:339–342.

[4] Gillespie, DT (2007) Stochastic simulation of chemical kinetics. *Annu Rev Phys Chem*.