Adaptation in Enzyme Networks: Searching for Minimal Mechanisms

Merrill E. Asp

A senior thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Bachelor of Science

Mark K. Transtrum, Advisor

Department of Physics and Astronomy

Brigham Young University

April 2016

ABSTRACT

Adaptation in Enzyme Networks: Searching for Minimal Mechanisms

Merrill E. Asp
Department of Physics and Astronomy, BYU
Bachelor of Science

Adaptation is an important biological function that can be achieved through networks of enzyme reactions. These networks can be modeled by systems of coupled differential equations. There has been recent interest in identifying what aspect of a network allows it to achieve adaptation. We ask what design principles are necessary for a network to adapt to an external stimulus. We use an information geometric approach that begins with a fully connected network and uses automated model reduction to remove unnecessary combinations of components, effectively constructing and tuning the network to the simplest form that still can achieve adaptation. We interpret the simplified network and combinations of parameters that arise in our model reduction to identify minimal mechanisms of adaptation in enzyme networks, and we consider the applications of these methods to other fields.

Keywords: Enzyme network, model reduction, adaptation, systems biology, information geometry, manifold boundary approximation method

ACKNOWLEDGMENTS

I would like to acknowledge the help of my wife Chelsea for her support and interest in my research, as well as Dane Bjork, who wrote code that greatly aided the programming necessary for this project. The computational work done here could not have occurred without the generous resources provided by the BYU Physics Department for my research group. Thanks also to all those who took the time to review drafts of this thesis, which greatly improved the structure and flow of the ideas presented here.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Enzyme interactions drive many interesting biological behaviors, and are thus scientifically valuable to understand (Ma et al. 2009). Enzyme reactions are a fundamental building block of biochemical pathways, and are among the most basic elements of the chemistry of life. The organizations of enzyme reactions determine in a non-trivial way what higher-level behaviors will emerge in the organism (Alon 2006). Networks of such reactions are studied in many contexts including gene transcription, metabolism, neurology, and cancer research. Continuing progress in these fields requires a strong understanding of how enzyme networks behave the way they do, and what fundamental aspects of a network are behind its specific behaviors.

To discover what minimal mechanisms are at play behind a certain behavior, we can first abstract a model of the behavior, often in the form of a network, with individual parts influencing each other in specific ways. From these networks we then seek equivalent, simplified networks in order to get a closer understanding of the behaviors they achieve. The function of a network can be understood on different scales of detail, from a black box interpretation of the network's

overall function to a detailed account of the individual molecular reactions that take place. A highly detailed, or fine-grained, model of an enzyme network is often too complex to provide a workable explanation for how a certain behavior occurs. However, some specific groups of details can be ignored or approximated, such as chemical parameters that do not individually impact overall behavior greatly, or subnetworks that have a single function. If these simplifications can be performed without losing the network's overall behavior, something valuable has occurred: The simpler model more clearly explains why a network performs its behavior, and is thus more useful for answering the questions of biological research, such as which network elements need to be corrected in case of an overall network malfunction. This application of model reduction is especially applicable to cancer research.

The question of how to properly simplify a network so that it still exhibits its desired behavior is generally difficult and has historically depended on experts applying hard-won intuition. The inherent nonlinearities of network behavior make straightforward attempts at model simplification difficult. One can always proceed on a case-by-case basis, but at the cost of time and generality. The method of discovering the principles that underlie one network behavior can be entirely unrelated to the methods necessary for another.

We show that identifying minimal mechanisms for the behavior of adaptation can be done systematically and in a way that scales well with system size, using information geometry to do so. We consider what is meant by the term "minimal mechanism," and clarify an objective definition in terms of our methods. Our results agree with and shed light on previous research and point out a way to study other behaviors and networks in other contexts.

1.2 Enzyme Networks and Systems Biology

Enzyme networks are graphically represented by nodes and edges, such as those in Fig. 1.2. Each node represents a fixed concentration of enzyme that can be either active or inactive in varying proportions, from 0% active to 100% active. Each edge is directional, and signals one node to either increase (promote) or decrease (inhibit) the fraction of active enzyme in another node. The more active enzyme is present in a node, the more edges proceeding from it will promote or inhibit other nodes.

We model nodes as dynamic variables in a system of coupled ordinary differential equations. These variables measure the fraction of active enzyme as a number from zero to one. Each edge is modeled as a term in one of the differential equations. The edges (or interactions between nodes) obey Michaelis-Menten rate equations, in accordance with other studies into enzyme network behaviors (Ma et al. 2009). The term representing one node (X_1) promoting another (X_2) is

$$\frac{dX_2}{dt} = C_{12}\frac{X_1(1-X_2)}{1-X_2+K_{12}}, \tag{1.1}$$

and the term for one node (X_1) inhibiting another (X_2) is

$$\frac{dX_2}{dt} = -C'_{12}\frac{X_1X_2}{X_2+K'_{12}}, \tag{1.2}$$

where we see each edge being characterized by two non-negative parameters, a strength parameter C and a shape parameter K. In the literature, these constants are called the catalytic rate constants and the Michaelis-Menten constants, respectively. Significantly, *both* network topology (how the network is connected) and these parameter values determine the behavior of the network. Because choice of parameters affects network behavior in a non-trivial way, the function of a network is an emergent property of both parameter values and network topology. This behavior can be calculated by numerically solving the coupled ordinary differential equations that model it.

The study of quantitative models of emergent biological behavior in networks is called systems biology (Alon 2006). Systems biology seeks to answer the question of what solutions nature can

produce to solve specific biological problems. To begin to tackle this question, a researcher must choose where to first look for the decisive aspects of a network that define its overall behavior. Literature on systems biology focuses on network topology, or the way the nodes of the network are connected, to explain how a network behaves a certain way. This is understandable because diagrams of network topology, such as the ones in Fig. 1.2, seem to hold intuitive explanations for how the individual enzyme reactions combine into a single output. However, it is already well-known that the network topology does not fully specify the network's final behavior, or even the function of the network's edges (Alon 2006). Often, specific equations with finely tuned parameters are necessary to fully specify a network. This disconnect between information and intuition motivates a different search for where the decisive mechanisms of network behaviors exist.

We intend to show that fundamental biological networks that achieve certain functions can be found and understood with a method of systematic model reduction described in Section 2.

1.3 Adaptation

We choose a specific behavior of enzyme networks to investigate. Adaptation, or the ability for a system to respond to a stimulus and afterwards return to equilibrium, is a significant behavior of enzyme networks (Artyukhin et al. 2009). This is the behavior we seek to understand in this thesis. Adaptation curves can be characterized by four parameters, as shown in Fig. 1.1.

We model adaptation as arising from a disturbance in a fully inactive enzyme network. This disturbance is a steady promotion of an input node that begins at a time $t = 0$. We choose an output node to exhibit the desired behavior. Any other nodes perform secondary tasks in order to translate the stimulus in the input node into adaptation in the output node. A typical diagram of this setup is shown in Fig 2.1b. There is general academic consensus that a network requires at least three nodes to achieve adaptation (Artyukhin et al. 2009). We thus narrow our search for minimal mechanisms

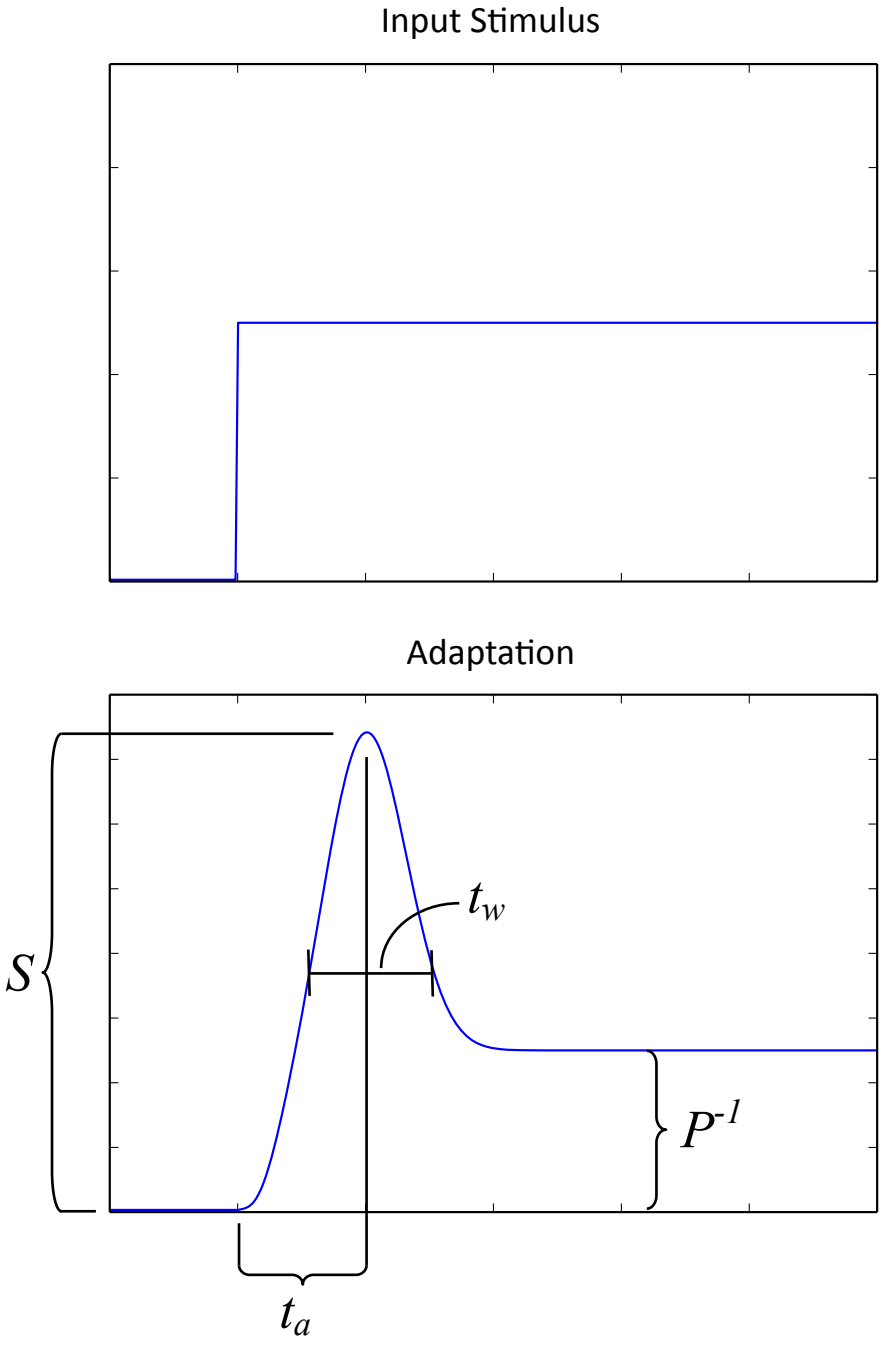

Figure 1.1 A typical adaptation curve (beginning from complete inactivity) with the parameters that describe its shape. This curve exhibits adaptation to the input stimulus that begins at time $t = 0$. The difference between the initial concentration and the maximum response is called the sensitivity (S). The inverse of the difference between the initial concentration and the final equilibrium is called the precision (P). There are also two characteristic times, the activation time (t_a), which elapses between the initial response and the maximum reponse, and the time width (t_w), which characterizes the width of the curve. Adapted from Ma et al. 2009.

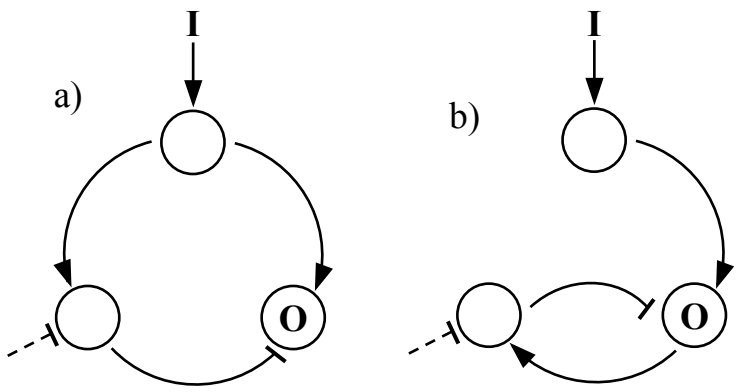

Figure 1.2 Two simple networks shown to achieve adaptation in previous research (Ma et al. 2009). Diagram (a) is an example of an incoherent feed-forward loop, and diagram (b) is an example of a negative feedback loop. Each circle represents an enzyme concentration that can be 0% to 100% active. The networks react to a steady input stimulus (I) that begins at a time $t = 0$, and the enzyme concentration in the output node (O) is checked for adaptive behavior. Each edge with an arrow represents active enzyme activating another node (promotion). Each edge with a bar represents active enzyme deactivating another node (inhibition).

of adaptation to three node networks, with one input, one output, and one secondary node.

Systems biology literature identifies two classes of network topologies that minimally achieve biological adaptation through a brute force search of three-node, three-edge networks (Ma et al. 2009). They are called the incoherent feed-forward loop and the negative feedback loop (see Fig. 1.2). These two classes of topologies are believed to be fundamental to adaptation because in a wider search of all three-node networks (with any number of edges), no adaptive network was found that does not contain one or the other. Although such methods can effectively identify simple networks that achieve adaptation, they are limited in their scope to very small networks, as will be elaborated on in Section 2.1. They also require extensive searches of an infinite parameter space, which can be difficult to declare exhaustive.

We seek a general method for relating behavior to mechanism that scales better with system size and provides insight into the way the network achieves adaptation.

Chapter 2

Methods

2.1 The Fully-Connected Network

The fully-connected network (FCN) contains three nodes and all possible activation and inhibition edges between the nodes, including inhibition from the environment (see Fig. 2.1a). Although this particular network is probably not present in nature and is certainly not minimal for adaptation, studying it can reveal information about minimal mechanisms and the relationships between them. This is because all three-node networks are contained in the FCN as special cases (as illustrated in Fig. 2.1). Since each edge of an enzyme network is modeled by a term in a differential equation, and each such term has a strength parameter, setting the strength parameter to zero is equivalent to removing the edge. The behavior of all three-node networks — including those that minimally achieve adaptation — is therefore somewhere within the FCN, for the right choice of parameters.

As further elaborated on in Section 2.3, we can fit the parameters of a given network to achieve adaptation. Performing model reduction on an FCN from many such initial sets of fitted parameters (fitting to adaptation along the way) is a highly scalable method for finding minimal mechanisms. An alternative method to studying the FCN is first tabulating all possible network topologies, and

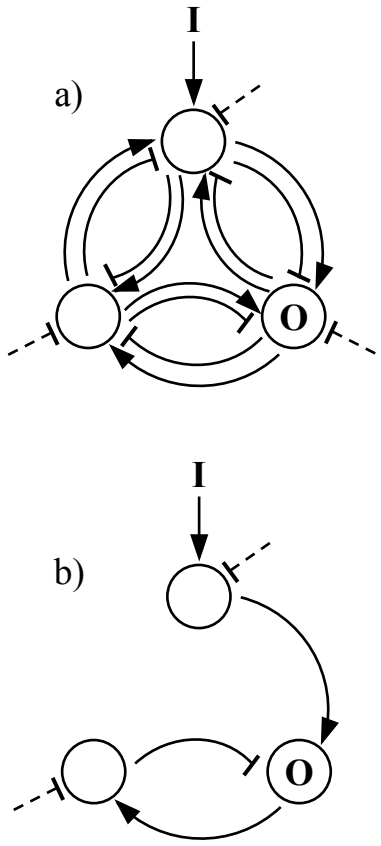

Figure 2.1 Diagram of the three-node fully connected network (a) with a special case (b), a negative feedback network. Each circle is an enzyme concentration that can be 0% to 100% active. The networks react to a steady input stimulus (I) that begins at a time $t = 0$, and the enzyme concentration in the output node (O) is checked for adaptive behavior. Each edge with an arrow represents active enzyme activating another node (promotion). Each edge with a bar represents active enzyme deactivating another node (inhibition). Each edge is defined by terms as Eq. (1.1) or Eq. (1.2) in a differential equation, with two parameters per edge. If the strength parameter in an edge of the FCN is set to zero, the edge is effectively erased. Erasing the appropriate edges from the FCN yields any particular three-node network, including the negative feedback network (b), which can achieve adaptation for the correct values of the remaining parameters.

then searching the parameter space of each such network one by one. This method has been used by other researchers (Ma et al. 2009). However, as these researchers claim, there are 16 038 network topologies for three nodes alone, and about 500 topologies were considered candidates for minimal networks (because they have only three edges). Each of these networks was tested with 10 000 different random parameter combinations to in order to find which networks could achieve adaptation. On the other hand, we search only one network topology, the FCN. It contains 31 parameters (the full network definition is contained in Appendix C), and finding minimal networks is achieved by performing model reduction repeatedly on the FCN until no more reductions can be performed. The difference between these approaches is illustrated in Fig. 2.2. In particular, we note the difference between the two definitions of a minimal network: the work of Ma et al. identifies three-edge adaptive networks as minimal, since no two-edge or two-node networks can adapt at all. The definition of a minimal network in this thesis is based on our model reduction process, and will be explained in further detail in Subsection 2.3.6.

Each such process of reductions (described in the rest of this chapter) produces a minimal network with no need for brute force searching an infinite parameter space or enumerating possible network topologies. Granted, the process of model reduction must be repeated from a different set of initial parameters to find another minimal network, but each such process yields a minimal network, which greatly improves efficiency of computation. The number of possible network topologies increases exponentially with number of nodes, with almost 500 000 possible four-node networks. In constrast, the number of parameters in an FCN increases only quadratically, with 57 parameters describing a four-node FCN.

Because of the inherent scalability of this method, model reduction from an FCN could be used to find minimal mechanisms that achieve behaviors more complex than adaptation that require more than three nodes.

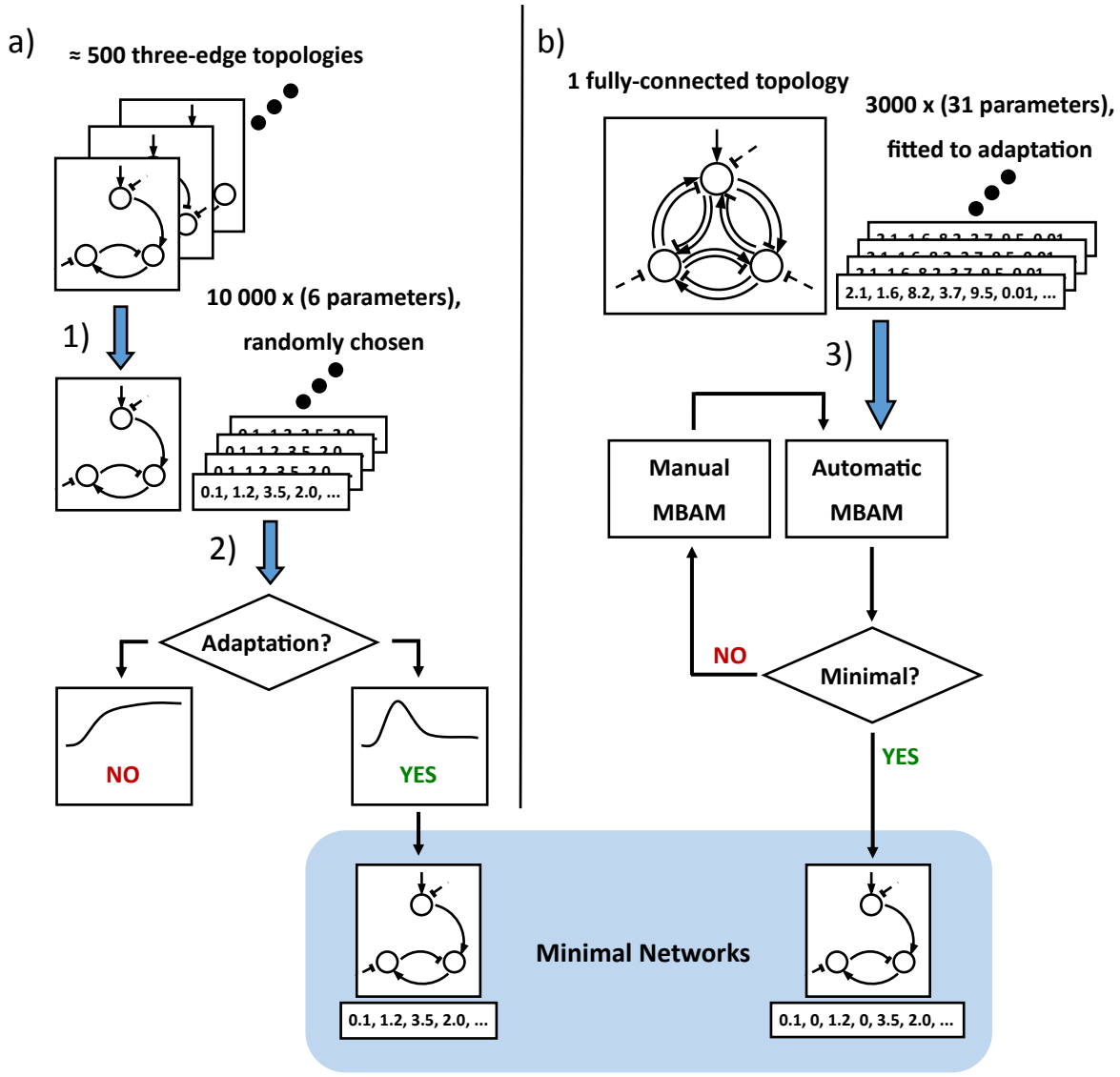

Figure 2.2 Alternative methods for finding minimal adaptive networks. This diagram contrasts the method of Ma et al. (a) with the method of this research (b), the manifold boundary approximation method (MBAM) from a fully-connected network (FCN). In (1), one network topology is chosen from about 500 choices (each which have only three edges). To this network is applied a group of 10 000 random parameter sets, each of which mathematically define the network's edges. In (2), each such set of parameters is used to find the output of the network, which may or may not exhibit adaptation. If the network is adaptive, it has been identified as a minimal adaptive network. In (3), a set of parameters for which the FCN is already adaptive is used to begin model reduction (explained in Section 2.3). Each such set of parameters can eventually produce a minimal adaptive network.

2.2 The Model Manifold

Enzyme network behaviors can be understood in terms of input and output. For our input, we choose a set of parameters to fill in our network definition and make it numerically explicit. Then we computationally solve the coupled differential equations of our network. This produces the network's output: a time evolution of the enzyme concentrations in the output node, such as that shown in Fig. 2.3. For our analysis, an enzyme network is a machine that takes in parameter values and outputs time evolutions. The model manifold is a means of representing all possible mappings from input to output.

The finite number of parameter values for a given model can be considered a vector. This vector indicates a position in parameter space. When we list the network's corresponding output at a finite number of time points, we obtain an output vector in data space. If we sweep through all parameter space vectors, we can sweep through all possible associated outputs, and we have traced out a smooth surface — a manifold — in data space. This manifold is called the model manifold, and it therefore contains all possible outputs of an enzyme network. This mapping from input (parameters) to output (time evolution) is elaborated on in Fig. 2.4, where we show a model manifold that can be visualized.

A specific network behavior (time evolution) is represented by a single point in data space. We create a toy function (see Fig. 2.3) to represent a typical adaptation curve. We sample 250 time points of this toy function, and we now have a vector in data space, which we call the adaptation point. Networks that achieve adaptation are represented by model manifolds that contain the adaptation point somewhere on their surface. This is to say that for some parameter values, that network can output an adaptation curve. It is possible that a given output (represented as a data space vector) can have many parameter space vectors that map to it or very close to it. The vectors in parameter space that map to the adaptation point indicate what parameter values such a model must have in order for adaptation to be achieved.

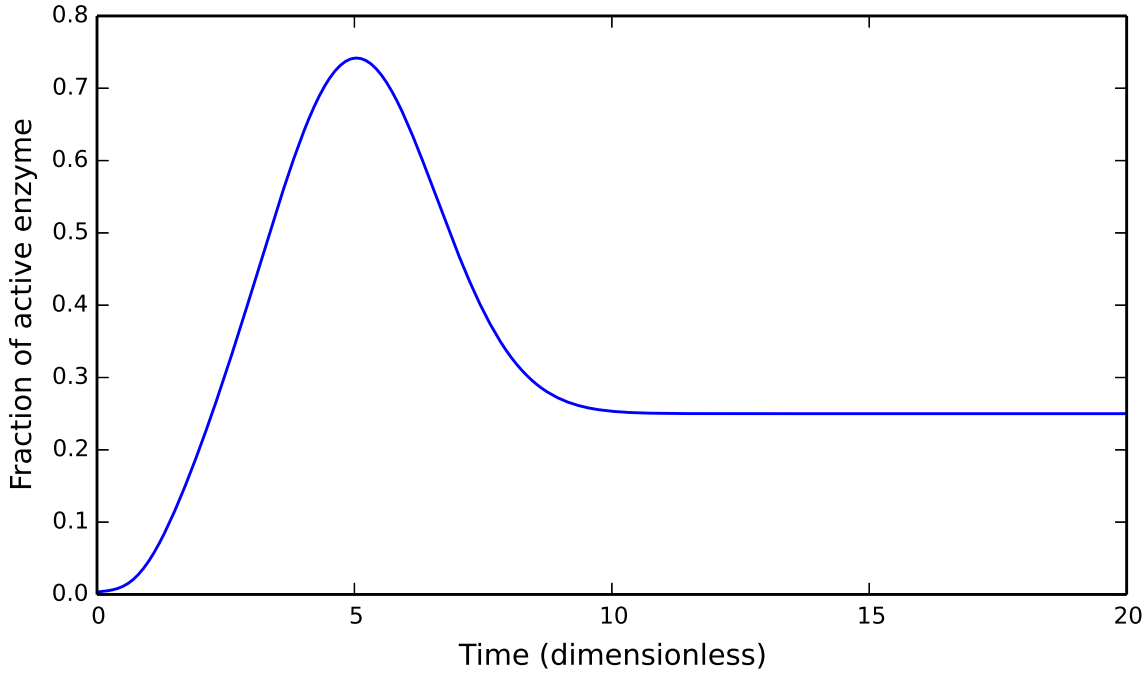

Figure 2.3 An enzyme network time evolution that exhibits adaptation. When the equations that define an enzyme network are solved, a function is produced that shows how the concetration of active enzyme in the output node changes with time. If that output resembles the function pictured, we say that the network is adaptive. This pictured curve is a toy function representing an ideal adaptation. Explicitly, the function is given by $\frac{1}{2}e^{-0.2(t-5)^2} + \frac{1}{4}(1-e^{-t})^5$, although actual time evolutions tend not to have simple analytic forms.

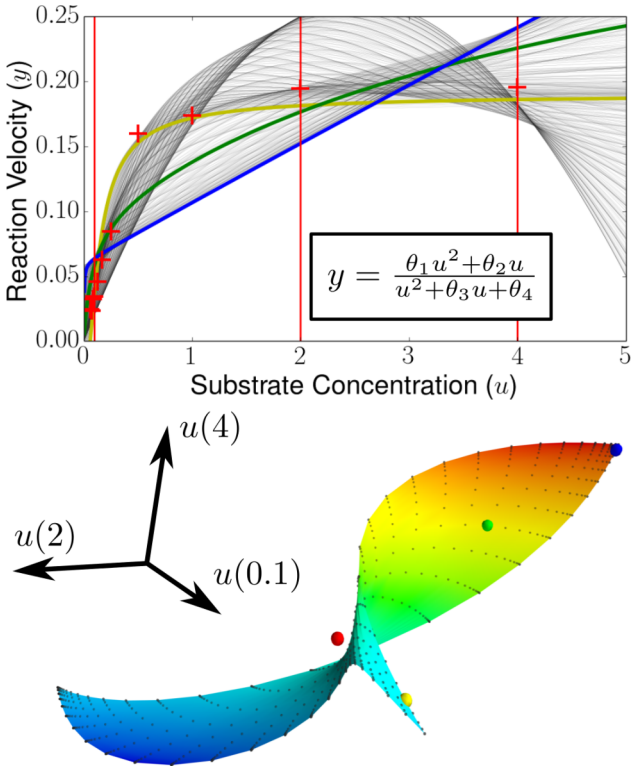

Figure 2.4 An example model manifold, showing the mapping from parameter space to data space. The function $y(u)$, which models enzyme reaction velocities, is written with its parameters θ_i. A target set of data points (red crosses) is graphed along with several curves representing $y(u)$ for different values of θ_3 and θ_4, with the other parameters held fixed. For one set of parameters, we obtain the blue curve, for another the green, and for another the yellow. We sample the y values of these curves at $u = 0.1, 2$, and 4, and each curve is mapped to a point on the smooth model manifold. Each axis of the data space in which the manifold is embedded corresponds to a different u-value, so each curve is mapped to a point in this three-dimensional data space. The blue curve is mapped to the blue dot, the yellow curve to the yellow dot, and so on, with the red dot representing the set of target data points (red crosses). The rest of the manifold is obtained by moving through all possible values of θ_3 and θ_4, sampling the y values, and sweeping out the surface in data space. Figure used with permission (Transtrum et al. 2015).

The number of dimensions in parameter space is the number of parameters in the network (N), and the number of dimensions in data space is the number of time points (M) chosen for the numerical output of the network. The model manifold is thus an N-dimensional surface embedded in an M-dimensional space. This manifold in its entirety is difficult to work with and impossible to visualize directly, since we choose a 31-dimensional manifold (31 parameters for the three-node FCN) embedded in a 250-dimensional data space (250 time points for the toy adaptation function). However, it is possible to find within this model manifold simpler models that still achieve adaptation.

2.3 Model Reduction

Model reduction is the process of finding a simplified, approximate model that still achieves a desired behavior. We perform model reduction by reducing the number of parameters one by one. The number of parameters required to characterize an adaptation curve (four, see Fig. 1.1) gives an estimate for the number of parameters the simplest adaptive network should have. We seek to reduce the FCN from its 31 parameters to about four in order to find the simplest mechanisms for adaptation.

We use the manifold boundary approximation method (MBAM) (Transtrum & Qiu 2014) to systematically reduce the FCN. This method relies on the fact that the boundaries of an N-dimensional model manifold are themselves $(N-1)$-dimensional model manifolds. These boundaries thus represent models with one fewer parameter. We therefore seek a trajectory (we choose a geodesic, as explained in Subsection 2.3.2) across the manifold that can take us to a boundary, and then the boundaries of that boundary, over and over. Hopping across the model manifold to regions of lower and lower dimension is the essence of the process of model reduction, sketched in Fig 2.5.

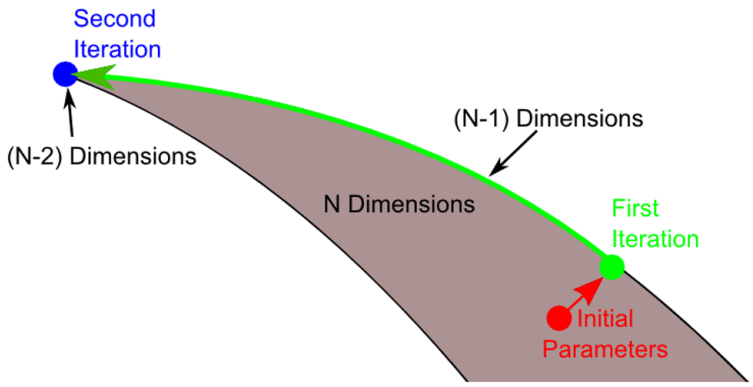

Figure 2.5 A diagram of model reduction cast as finding manifold boundaries. From a point on the manifold determined by initial parameters, a geodesic curve can be drawn along the N-dimensional manifold (N-parameter model) to its boundary (an $(N-1)$-parameter model). Within this boundary, another boundary can be found, recursively finding simplified models with one fewer parameter each step. As long as this process stays within the vicinity of the adaptation point, all of the reduced models will be able to achieve adaptation. Figure used with permission (Transtrum & Qiu 2014).

The full process of MBAM can be expressed algorithmically, as follows:

1. Gather a large sampling of initial parameter values that achieve adaptation in the FCN.
2. From an initial point, calculate a geodesic across the model manifold that stays near the adaptation point.
3. Identify when the geodesic is approaching a boundary.
4. Interpret the boundary as a limit of the model parameters.
5. Apply that limit to the equations of the model to create a simplified model.
6. Find a parameter vector that achieves adaptation in the new model.
7. With that parameter vector as jumping-off point, repeat the process from step 2 until the simplified model can no longer achieve adaptation.

This process terminates in a maximally simplified model with as few parameters as possible to achieve a desired behavior. If it is then repeated for each of the initial points in the large sampling, we can be confident that a thorough search for minimal networks has been accomplished.

This overall process is scalable, deterministic, and general for many complex systems that can be modeled with coupled ordinary differential equations. We will interpret the minimal models produced through MBAM to identify the fundamental mechanisms of adaptation in Section 3. For now, we approach each step of the overall process in more detail.

2.3.1 Initial Points

Because the coupled equations of the FCN are non-linear, we must use a specialized method to find parameter values that can achieve adaptation. The Levenberg-Marquardt algorithm is used, from a large number of randomly distributed points in parameter space, to robustly converge on parameter values that achieve adaptation in the FCN. The algorithm was designed to solve just such a parameter optimization problem as this one, for which there is a target function that a parameter-based model is to achieve (Marquardt 1963). We choose our target function to be the toy function chosen previously (see Fig. 2.3), although any function can be used in principle. Roughly one third of the random points converge to succesful initial points that produce output almost identical to the toy funtion. Model reduction can begin from these points.

2.3.2 Calculating the Geodesic

An initial point in parameter space corresponds to a particular point on the model manifold. From this point, we calculate a geodesic across the model manifold. Geodesics are paths of least distance on a curved surface, and can thus be used to approach boundaries as quickly as possible, without the meandering that a non-geodesic path may take across the manifold.

A geodesic is determined by the second-order differential equation

$$\frac{d^2x^k}{ds^2} + \Gamma^k_{ij}\frac{dx^i}{ds}\frac{dx^j}{ds} = 0 \tag{2.1}$$

in the parameter space position x^i (the i is a superscript, not an exponent, which labels the compo-

nent of the vector x) with s a distance parameter in data space along the geodesic and the Christoffel symbol Γ^k_{ij} containing curvature information about the mapping between parameter space and data space. This equation can be reliably solved by a standard numerical differential equation solver. The calculation of a geodesic reveals a path in data space by determining the path in parameter space that corresponds to it. Because the geodesic equation Eq. (2.1) is second order, the path of a geodesic is fully specified by an initial point and an initial direction in parameter space. We have our initial point, so we now ask which initial direction should be chosen in order to reliably find a useful boundary.

A natural choice for this direction comes from the singular value decomposition of the Jacobian matrix that describes the mapping from N-dimensional parameter space to M-dimensional data space at our initial point (Transtrum & Qiu 2014). Essentially, the Jacobian matrix J contains local information that maps a small change in parameter space, $\mathrm{d}x_i$, to the corresponding small change in data space, $\mathrm{d}y_i$ (we now use subscripts to label different vectors). This relation can be written as

$$J\,\mathrm{d}x_i = \mathrm{d}y_i. \tag{2.2}$$

The singular value decomposition of J can be written as

$$Jv_i = \sigma_i u_i, \tag{2.3}$$

where v_i is one of N unit vectors in $\mathbb{R}^N$, u_i is one of N unit vectors in $\mathbb{R}^M$, and the σ_i are non-negative numbers called the singular values. Comparing Eq. (2.2) to Eq. (2.3), we see that v_i corresponds to a direction in parameter space that maps to the direction u_i in data space, scaled by a factor of σ_i. Typical values of σ_N, the smallest σ_i, can be as small as 10^{-14} in the FCN. When we calculate the singular value decomposition of J and thus find the vector v_N corresponding to this smallest singular value, we have found a direction in parameter space that causes an almost imperceptible change in data space (the output of our model). This is the parameter space direction in which we choose to start our geodesic, so that we do not stray far from our desired output.

Unfortunately, the calculation of the vector v_N is not unique, and may yield either v_N or $-v_N$, so we must have one extra criterion for choosing whether to pursue the v_N direction forwards or backwards. We choose the direction that causes the parameter velocities to initially increase. The motivation for this criterion is described in the next subsection.

2.3.3 Identifying Boundaries

In order to determine if a geodesic has run sufficiently close into a boundary on the model manifold to be able to discern what the boundary is, we may track the change in parameter values as we move along the geodesic. As the geodesic comes steadily closer to a boundary that is defined by some limit in the parameter values, the parameter values themselves will asymptotically approach that limit (see Fig 2.6). For example, if a manifold boundary is defined by the parameters C_{13} and K_{13} approaching infinity, as the geodesic approaches the boundary in data space, the parameters will skyrocket in magnitude as they race to keep up. This is evident in the parameter velocities, or the rate of change of the parameters with respect to motion along the geodesic. These velocities grow very large when a manifold boundary begins to be approached.

A significant increase in parameter velocity (typically 25 times larger than the initial velocity) thus signals when a boundary has been reached. This also gives us our desired criterion for choosing either v_N or $-v_N$ for our initial geodesic direction: choose the direction that initially causes the parameter velocity to increase rather than decrease, which generally indicates continued parameter acceleration afterwards.

2.3.4 Interpreting Boundaries

It is helpful at this point to reconsider why locating and interpreting boundaries of the model manifold is a method of model reduction. The FCN is represented by a 31-dimensional model manifold, so the boundaries of this manifold are 30-dimensional — just as the boundary of a two-

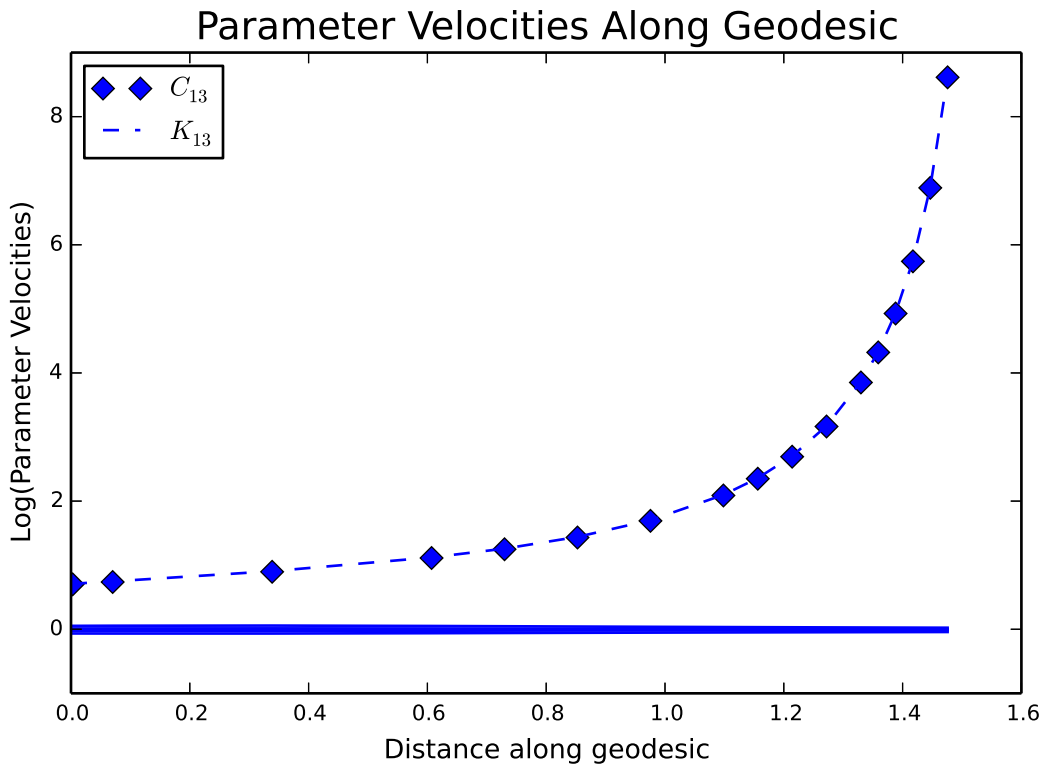

Figure 2.6 Semilogarithmic plot of parameter velocities along a geodesic, with one curve for each parameter. This model has 14 parameters, but only the curves for the two parameters C_{13} and K_{13} are labelled, since the other parameter velocities are indistinguishable from each other. This clearly indicates the limit $C_{13}, K_{13} \to \infty$. The significant increase in overall velocity also indicates that the boundary corresponding to this limit is being approached on the model manifold. Parameter values can also be considered for discerning a limit. Compare Fig. 2.7.

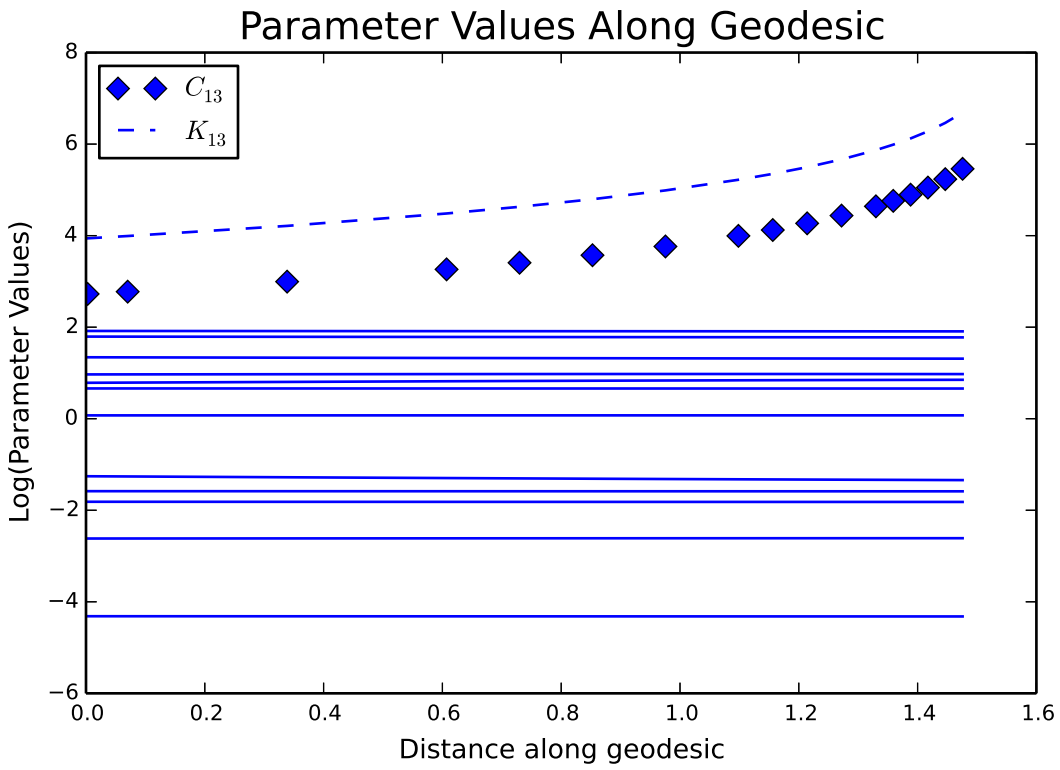

Figure 2.7 Semilogarithmic plot of parameter values along the same geodesic as used in Fig. 2.6, with one curve for each parameter. This model has 14 parameters, but only the curves for the two relevant parameters C_{13} and K_{13} are labelled. This indicates the limit $C_{13}, K_{13} \to \infty$, where the steady ratio between the two parameters (a steady difference on this plot) is visible. Parameter velocity plots such as those in Fig. 2.6 are often used to discern parameter limits, since they contain the same limit information as parameter value plots in a clearer form.

dimensional manifold (such as a disk) is one-dimensional (the circle bounding the disk). These boundaries are smooth, finite manifolds themselves, but of a lower dimension. They also represent enzyme networks that achieve adaptation, but that do so with one fewer parameter. Fortunately, the models that these simpler manifolds represent can be found from the FCN by interpreting a given boundary, as I now explain below.

Once it is clear that our geodesic has brought us near an edge on the model manifold, we can see how the parameters change along the geodesic. There are two particular limits that are easily discerned by looking at the parameter velocities along the geodesic, one of which is shown in Fig. 2.6. In fact, these limits can be recognized automatically by a computer, as will be expounded upon in Section 2.5. They consist of one parameter approaching zero, or two parameters approaching infinity at the same rate. These limits are indeed extreme parameter values, which result in outputs of the model that cannot be exceeded for any other parameter combination. This is why we can interpret model manifold boundaries as limits of the model parameters.

2.3.5 Simplifying the Model

When parameter limits are applied mathematically to the equations of the model, a simplified model emerges with one fewer parameter. The two most common limits and their effects on the terms of the model's differential equations are tabulated in Table 2.1. We note here that the work of Ma et al. identified these limits (called saturated and linear, respectively) in order to specify the parameter ranges in which their classes of minimal networks (see Fig 1.2) achieve adaptation.

In general, there are many different types of limits that can be discerned from parameter velocities along a geodesic, each one producing one fewer parameter in total when applied to a model's differential equations. New parameters may arise that are combinations of the old parameters. The form of the equations may also change significantly (as seen in Table 2.1). It is in these emergent parameters and the new forms of the model's equations that minimal mechanisms begin to reveal

Table 2.1 Common parameter limits for Michaelis-Menten dynamic equations. As parameters in a model reach certain extreme values, their limits can be recognized, and these limits can be applied to the model's equations to create simpler equations, and thus a simpler model. These limits often affect only a few terms of the full equations, so we have included just one term to demonstrate how the model's equations change. For these expressions, the X_i are dynamic variables measuring the fraction of active enzyme in node i, and all other variables are model parameters.

Parameter limit	Term before limit	Term after limit[a]	New parameters
$K \to 0$	$\frac{CX_1(1-X_2)}{1-X_2+K}$	$CX_1\mathscr{H}(1-X_2)$	None
$C,K \to \infty$	$\frac{CX_1(1-X_2)}{1-X_2+K}$	$pX_1(1-X_2)$	$p = C/K$

[a] The function $\mathscr{H}$ used here is the Heaviside step function, with convention $\mathscr{H}(0) = 0$.

themselves.

We note here that when an edge of the FCN is "erased," the C and K parameters of the edge are usually combined first, and the combined parameter C/K then approaches zero, so that only one parameter is removed at a time. In general, parameter limits either remove edges from a network, or they fix ranges in which edge parameters must fall in order for the network to achieve adaptation. We consider this interpretation of parameter limits further in Section 3.1.

If the geodesic has not taken us far from the adaptation point on the model manifold, we can be confident that the manifold boundary in which we now reside contains or is very near the adaptation point. Thus the simplified model we have found can likewise achieve adaptation.

2.3.6 Fitting the New Model

In general, a geodesic traced across the manifold will terminate on a point in data space that is not as close to the adaptation point as possible. This requires us, once we have written a new reduced model, to do two things: First, we must find the $N-1$ parameters that correspond to the end of the

geodesic in our new model, and then we must perform another Levenberg-Marquardt fitting to find the parameter vector that brings the output of the reduced model as close to the adaptation point as possible.

We can take the N parameters of the parameter vector at the end of our geodesic, and apply the limit we discerned to find $N-1$ parameters that correspond to an equivalent point on the manifold boundary, the new model. After the Levenberg-Marquardt fitting, we have a parameter vector that achieves adaptation in the model with $N-1$ parameters.

This simplified model defines a new manifold, and we have a new initial point to run geodesics from. The algorithm can be repeated identically from this new initial point, finding boundary within boundary and interpreting limit after limit until there is no manifold boundary that contains or is nearby the adaptation point.

Once a manifold contains the adaptation point but none of its boundaries do, we can say that the model corresponding to that manifold is fully reduced. This also serves as our definition of a minimal network. That is to say, when a network cannot be further reduced without losing its adaptive behavior, it is minimal. It is at this point that we can analyze the fully-reduced network equations and their emergent parameters (which are algebraic combinations of the original FCN parameters) to identify the minimal mechanisms that achieve adaptation.

2.4 Manually Discerning Limits

As explained in Section 2.3, the process of model reduction with MBAM involves calculating trajectories (specifically, geodesics) in parameter space. These trajectories are then inspected to see what limit of the parameters they represent, and then this limit is applied to the equations of the model to yield a simplified model. This process is repeated over and over again until the model cannot be simplified any further and still exhibit adaptation.

Although common, the easily recognizable limits like the ones in Table 2.1 are not the only limits to appear. It takes 27 model reduction steps to bring the 31-parameter FCN to the goal of four parameters, and it is likely that other, less clear limits will appear during that process. Typically, it is not difficult to identify a novel limit, since almost all limits are represented by parameters approaching zero or diverging to infintity (occasionally a limit will have two parameters becoming equal to each other, which may be more subtle to discern). However, applying a new type of limit to the equations of the model can be challenging.

The guidelines for applying limits are simple: Each parameter approaching a specific value must be removed from the model. If P parameters are involved in the limit, a total of $P-1$ new parameters must be introduced that are combinations of these parameters, bringing the net total number of parameters down by exactly one every model reduction step (Transtrum & Qiu 2014). New parameters must be finite, made in combinations such as ∞/∞, $0\cdot\infty$, $\infty-\infty$, or $0/0$.

Although the requirements for a correct reduction are simple, evaluating an unusual limit may involve multiple equations, require rescaling of the dynamic variables, or change a differential equation into an algebraic equation, as shown in the limits delineated in Appendix A.

We show an example of a less common limiting process. The affected equation is

$$\frac{dX_2}{dt} = -\theta_0 X_2 + C_{12} X_1 \mathscr{H}(1-X_2) + \frac{C_{32} X_3 (1-X_2)}{1-X_2+K_{32}} - C'_{32} X_3, \tag{2.4}$$

with θ_0 a parameter created in previous reductions, and the limit is

$$K_{32} \to 0,$$

$$C_{32},\, C'_{32} \to \infty.$$

We may group the last two terms of Eq. (2.4), writing $1 - X_2 = X_{2I}$ for brevity, and obtain

$$X_3 \left(\frac{C_{32} X_{2I}}{X_{2I} + K_{32}} - C'_{32} \right)$$

$$= X_3 \frac{C_{32} X_{2I} - C'_{32} X_{2I} - C'_{32} K_{32}}{X_{2I} + K_{32}}$$

$$= X_3 \frac{(C_{32} - C'_{32}) X_{2I} - C'_{32} K_{32}}{X_{2I} + K_{32}}.$$

We are now in a position to evaluate the limit, which we write into Eq. (2.4) to obtain

$$\frac{dX_2}{dt} = -\theta_0 X_2 + C_{12} X_1 \mathscr{H}(1 - X_2) + X_3 \frac{\theta_1 (1 - X_2) - \theta_2}{1 - X_2 + \varepsilon}. \tag{2.5}$$

We have included a small constant ε to keep the denominator of the final term nonzero when $X_2 = 1$, but this quantity is not a model parameter (meaning it is chosen arbitrarily, and it is not tuned to fit the reduced model to the adaptation point). Two new model parameters were obtained, defined by $\theta_1 = C_{32} - C'_{32}$ and $\theta_2 = C'_{32} K_{32}$, which are both finite combinations of the divergent parameters. These two new parameters offset the three parameters we have eliminated so that the net number of parameters decreases by exactly one. At this point, we accept Eq. (2.5) as the differential equation of our new model.

2.5 Automating Model Reduction

The algorithmic structure of MBAM lends itself to being implemented as a program with limited human intervention. One of the primary goals of this research has been the implementation of an automated model reduction program. This program is largely based upon functions in the Python programming language (using the NumPy and SciPy packages) organized into a class structure. Original code is included in Appendix B, with a flowchart detailing its structure in Fig. 2.8.

Reduction begins with a model definition file for the FCN (a file like `Enz3_31.py` in Appendix B). This file contains all of the parameter names and the explicit differential equations that

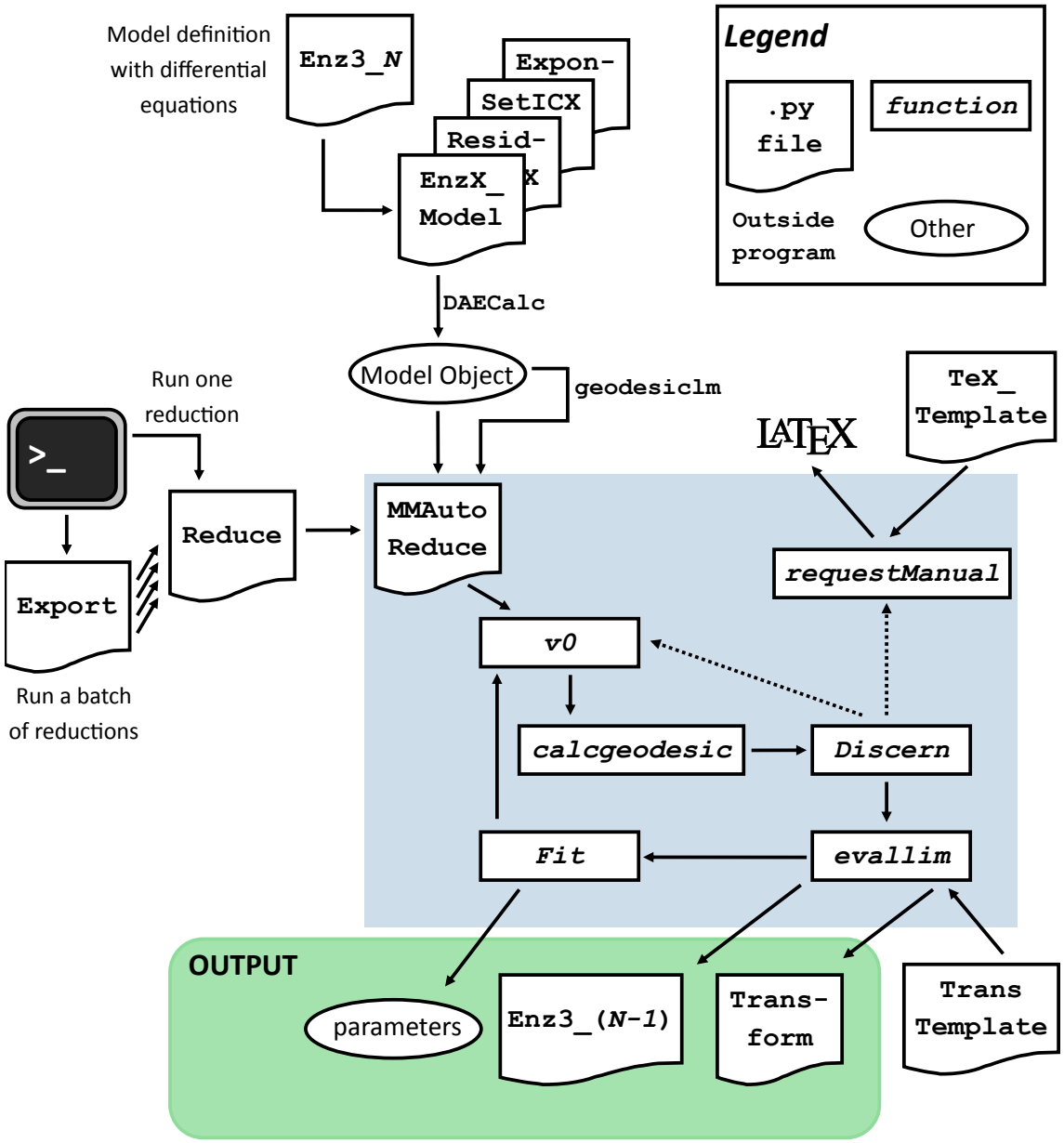

Figure 2.8 Flowchart for the automated model reduction code. A prepared model definition file (upper-left) can be used to begin a model reduction from the console (center-right). The MBAM algorithm is run by the functions of the `MMAutoReduce` class (blue box). This produces output (green box) as the algorithm iterates to a minimal network, or a LaTeX document specifying a limit for the user to evaluate manually.

define a three-parameter FCN. Using the `DAECalc` class written by Mark K. Transtrum, a suite of four Python scripts (including `EnzX_Model`) creates a model object, from which time evolutions, the Jacobian matrix, and other quantities can be accessed directly. It is at this point we choose adaptation as our goal behavior (the toy adaptation curve in Fig. 2.3 is coded into `ResidualsX.py`), as well as the initial conditions for our network (in `SetICX.py`). The file `Exponential.py` provides for the model object to work not with parameter values directly, but with the logarithm of the parameter values, also called log-parameters. This is appropriate, since all parameter values are positive, and can differ in size by many orders of magnitude.

This model object is used by the rest of the model reduction code. The function `geodesiclm` uses this model object to perform Levenberg-Marquardt fitting of the FCN, which gives us our initial parameters that achieve adaptation for the FCN. For each set of initial parameters, we have a folder that will contain all model reduction information and output from that parameter vector.

The user can begin a reduction by calling `Reduce.py` from the console (or `Export.py`, to run a batch of reductions simultaneously) when initial parameters and a model object have been prepared.

We now consider the automatic reduction algorithm itself, contained in `MMAutoReduce.py`, by touching on each function contained in the class definition.

2.5.1 Primary Functions

We begin with the first major function that `MMAutoReduce.py` calls after starting a new reduction from an initial point in parameter space.

`v0` — First, the Jacobian matrix is calculated at the initial point. As explained in Subsection 2.3.2, the singular value decomposition of the Jacobian matrix is used to choose an initial direction for the geodesic. This initial point and initial direction are passed to `calcgeodesic`.

`calcgeodesic` — A path in parameter space corresponding to a geodesic in data space is

calculated stepwise, at each point measuring the increase in parameter velocity. Once the velocity has increased by a factor of 25 from its initial size, the geodesic stops. Subsection 2.3.3 explains that this implies that the geodesic is near a manifold boundary, where a simplified model can be found. This new position in parameter space is passed to `Discern`.

`Discern` — The Jacobian matrix and its singular value decomposition are again calculated at the end of the geodesic in order to discern a parameter limit. The direction v_N, corresponding to the parameter space direction that causes the least change in the model's output (see Subsection 2.3.2), is checked for large components. These large components correspond to parameters that are diverging to infinity or to zero (since we use log parameters, diverging to a value of zero means diverging to negative infinity in log space, making these limits easier to see). The two types of limits mentioned in Table 2.1 can be discerned and evaluated automatically.

If a different type of limit is discerned, the reduction will first try to run `v0` again from the initial point, but in the reverse direction. Although the criterion for choosing the direction of the geodesic mentioned in Subsection 2.3.2 is often reliable, it is not infallible, and occasionally the boundary of the model manifold lies in the opposite direction of what was expected.

If the backwards running geodesic also fails, the function `requestManual` is run, which stops automatic reduction and produces a LaTeX document specifying what type of limit was discerned, so that a manual reduction can easily be begun.

However, if one of the two types of limits in Table 2.1 is discerned, the parameter limit information is passed to `evallim`.

`evallim` — A parameter limit is evaluated with an expanded version of the `symbolic` package, written by Dane Bjork using `SymPy`. A single parameter approaching zero can always be evaluated, simply be replacing the parameter with ε, an arbitrary small constant (which we fix at 10^{-4}). This prevents division by zero in our computational models. Incidentally, this ε is also used to make a differentiable analog to the Heaviside step function $\mathscr{H}(x) \approx x/(x+\varepsilon)$, for non-negative x. Also,

the function `check_ck` [1] is used to recognize if two parameters that diverge to infinity are part of the same term, in which case the limit is evaluated using the `gruntz` function in `SymPy`, which evaluates simple symbolic limits. These limits are evaluated in the model definition file, creating a new model definition file, such as `Enz3_30.py`, with one fewer parameter and simpler differential equations. The parameters of the new model are renamed to run from `p0` to `p(N-1)`, where N is the reduced number of model parameters. This is done for convenience in working with the parameters, which are internally numbered from zero to $N-1$. The new model definition file is then used to make a new model object (as `Enz3_31.py` was).

Information about which limit was discerned is further used in multiple places. First of all, a Transform file is written using the template `TransTemplate.py`. This Transform file algebraically relates the parameters of the new model to the parameters of the old model, allowing one to systematically retrieve what the combined parameters of the reduced model are in terms of the original FCN parameters. Second, the position of the end of the geodesic is still given by N numbers, instead of the $N-1$ numbers that should specify a position in the parameter space of the new reduced model. The Transform file converts the N parameters to an array of $N-1$ parameters, which is now passed to `Fit`.

`Fit` — As mentioned in Subsection 2.3.6, another fit with the Levenberg-Marquardt algorithm is necessary to ensure that the new model achieves adaptation as close as possible to the ideal time evolution shown in Fig. 2.3. Once this fit is completed, we now have a new initial point to calculate our geodesic from, and the process can repeat, by passing this new initial point to `v0`. This repetition is written into a for loop in `Reduce.py`. Assuming the previous reduction occurred without error, the next reduction step is taken automatically.

[1] written on page 74 in Appendix B

2.5.2 Effectiveness of Automation

In this way model reduction with MBAM can be automated. This process will terminate either with a minimal network or information on what parameter limit needs to be manually evaluated in order to proceed with model reduction, whereafter automatic reduction can begin again.

As an indicator of the effectiveness of the current version of the automatic reduction code, a representative sample of 72 reductions (performed from 72 distinct sets of parameters that produce adaptation in the FCN) was run automatically. The number of model reductions the program successfully performed before a manual reduction was required is organized in Fig. 2.9. In short, the number of automatic reductions ranged from 0 to 26, with about 15 being most common, and 11 the average. Thus several models were reduced just a few steps away from fully-reduced networks, which requires about 27 reduction steps. At the point each automatic reduction stopped, a manual reduction can be perfomed, and then automatic reduction can be resumed.

All reductions in the sample were performed simultaneously, and all had terminated after about three hours. In general, each reduction requires anywhere from one to twenty minutes, depending on the number of steps that need to be calculated for the geodesic.

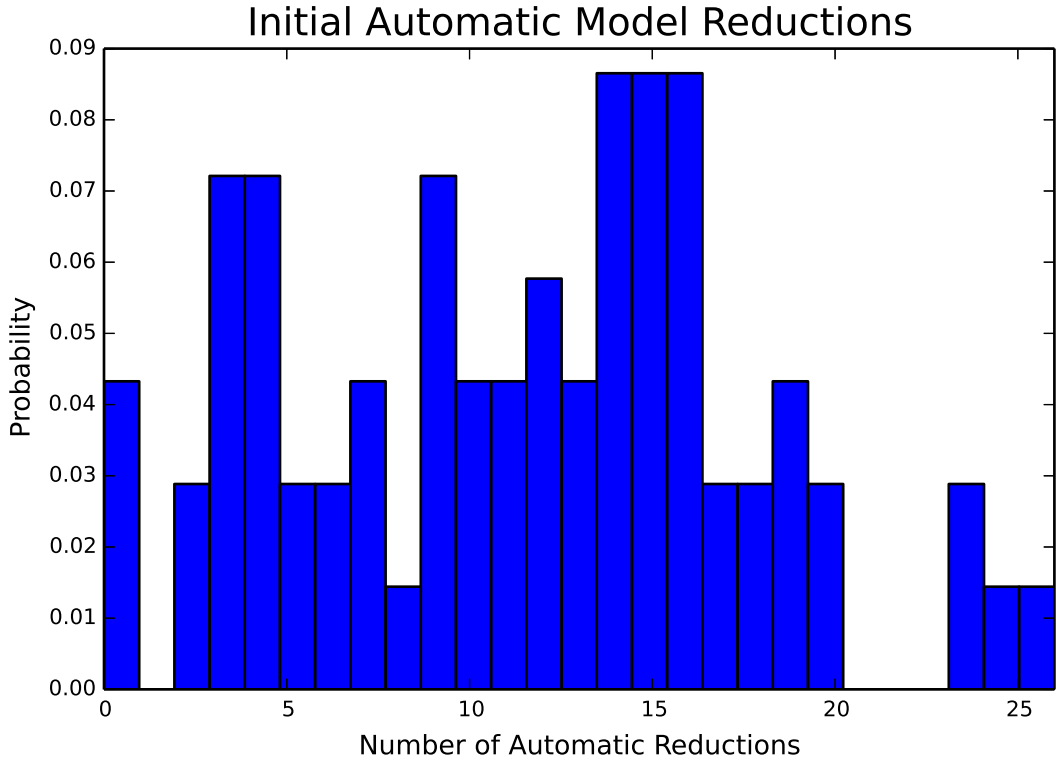

Figure 2.9 Probabilities of the number of initial automatic reductions performed before manual reduction was required. This data comes from a representative sample of 72 model reductions from the FCN.

Chapter 3

Results

3.1 Reduced Networks and Future Research

Using a combination of automated model reduction and manual limit interpretation, several reduced networks have been found. We consider one such reduced network as an informative example. Although it is not fully reduced according to the definition in Subsection 2.3.6, it has begun to show several of the features of a fully-reduced network, and we can use it to understand a minimal mechanism of adaptation. We include the network diagram in Fig. 3.1. This diagram can be produced either from inspection of the equations that define the model, or by going through the reduction steps individually and erasing edges from the FCN when the reduction step sets the strength parameter of the edge to zero. We will elaborate further on the relationship between the network diagram and the model definition equations. The equations for each node are reproduced here:

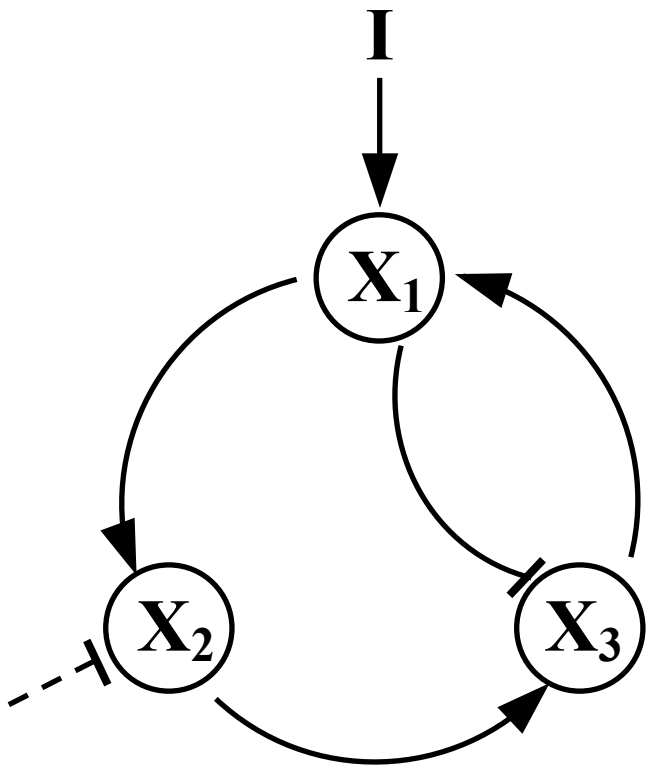

Figure 3.1 Network diagram for a reduced network, with steady input stimulus I beginning at time $t = 0$ and X_3 being the output node in which adaptation behavior is expected. This representation arises from the network definition equations Eq. (3.1) - (3.3). Equivalently, as the FCN was reduced to eventually obtain this network, the C/K ratios of various edges were set to zero one by one. Erasing these edges yields the diagram shown. The network achieves adaptation for the correct values of the network definition parameters.

$$\frac{dX_1}{dt} = I_{IC}\frac{1-X_1}{1-X_1+K_{IC}} + C_{31}X_3\mathscr{H}(1-X_1) \tag{3.1}$$

$$X_2 = \frac{\theta X_1}{1+\theta X_1} \tag{3.2}$$

$$\frac{dX_3}{dt} = C_{23}X_2\mathscr{H}(1-X_3) - C'_{13}\frac{X_1X_3}{X_3+K'_{13}}. \tag{3.3}$$

Two features of these equations are immediately of note — first that Eq. (3.2) is an algebraic equation instead of a differential equation, and second that there is an emergent parameter, θ, that has resulted from the reduction. The value of θ in terms of the FCN parameters is

$$\theta = \frac{C_{12}K'_{E2}}{K_{12}C'_{E2}}.$$

These two aspects of the reduced equations are linked — the equation for X_2 became algebraic in a reduction step in which both C_{12} and the ratio parameter C'_{E2}/K_{12} tended to infinity together[1]. Since C_{12} characterizes how strongly X_1 promotes X_2 and C'_{E2} characterizes how strongly X_2 is inhibited by the environment, we can interpret this limit as a balance being struck between the inhibition and promotion of X_2. Since both of these reactions are very strong, they occur with a very short reaction time, causing X_2 to be slaved to X_1 according to Eq. (3.2). Both reactions are present, which is why they are included on the network diagram in Fig. 3.1, even though their presence is not obvious from the appearance of Eq. (3.2).

However, in a very real sense, the model definition equations are a more precise way of articulating the network behavior than the network diagram. It is not at all obvious from the network diagram that there is a functional relationship between the concentrations of active enzyme in X_1 and X_2. Also, the form of the emergent parameter θ shows that the individual parameters C_{12}, K_{12}, K'_{E2}, and C'_{E2} are not significant individually, but only in combination. Adjusting the parameters one at a time yields complementary effects which may amplify each other or cancel each other out. For example, doubling both K'_{E2} and K_{12} would have no effect on the output of the reduced network. This fact is not apparent from the network diagram alone. The rules for these cancellations and amplifications are new information that we have obtained through the process of model reduction, and that information is encoded in the reduced equations.

[1] This limit is shown in Appendix A on page 42, with $\theta_0 = C'_{E2}/K_{12}$, although two more reduction steps are required before the model definition equation can be written as Eq. (3.2).

For this network, the essential minimal mechanism is a secondary node that channels the input stimulus directly into the output node, allowing the input and output node to directly balance each other. Notably, this network diagram does not fall into the two classes (see Fig. 1.2) believed to be the *only* basis for adaptation in enzyme networks. Although it seems to have a negative feedback loop built into it (compare Fig. 1.2b), X_2 being slaved to X_1 provides for X_1 to directly promote and directly inhibit X_3, instead of X_2 being used as a buffer, as the negative feedback loop does. It is likely because this network has four edges that it was not identified as minimal in the work of Ma et al. However, we can show that this network achieves adaptation with arbitrary sensitivity and precision[2], as shown in Fig 3.2. We have thus uncovered a new possible solution that nature has for the problem of adaptation.

We should remark that when we speak of a "minimal mechanism" for adaptation, the only way to be completely clear about what we mean is to specify the transformed network equations and emergent parameters of our network. The work of Ma et al. also included parameter limits for each minimal adaptive network, and it identified trends in parameter values that increase the likelihood of a given network topology being adaptive. These parameter ranges were pursued by intuition and mathematical consideration of individual networks. In contrast, MBAM automatically generates limits that indicate what parameter ranges are necessary for a network to adapt, decreasing the need for ad-hoc analysis and facilitating in-depth understanding of numerous adaptation mechanisms. With the many limits that arise from the Michaelis-Menten equations through MBAM, we have a much richer language at our disposal to express how a network achieves adaptation the way it does. The most significant classes of limits and several examples are included in Appendix A.

In short, the limits performed in the process of model reduction inform us which edges of the FCN can be ignored, which extreme values parameters must have for the network to properly

[2]For completeness, we add that the parameter values necessary to achieve the adaptation curve in Fig 3.2 with highest sensitivity and precision are $K_{IC} = 18.76$, $C_{31} = 0.23$, $\theta = 37.02$, $C_{23} = 2.06$, $C'_{13} = 22.56$, $K'_{13} = 0.25$.

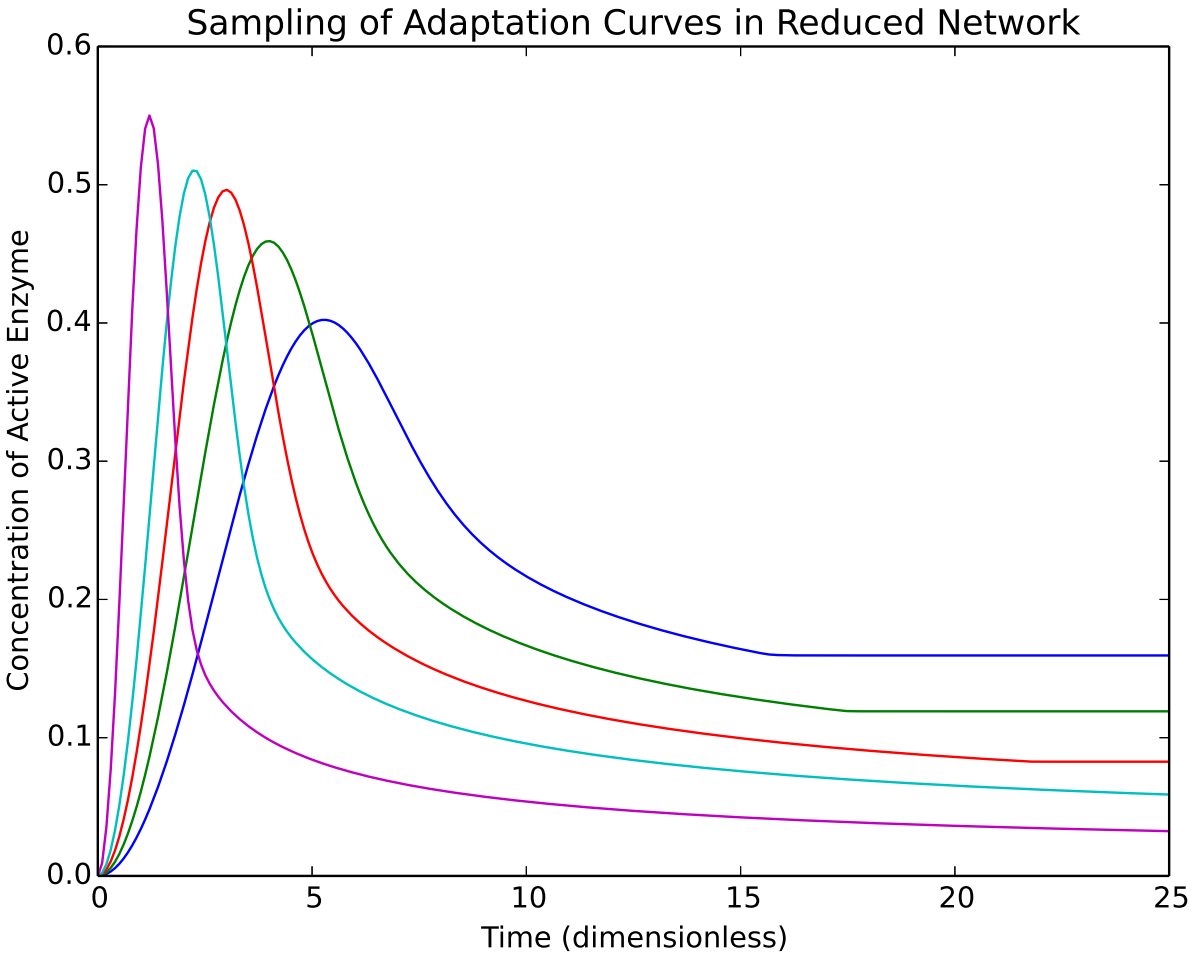

Figure 3.2 A sampling of output curves showing the variety of adaptation behaviors that the reduced network of Eq. (3.1) - (3.3) can produce. Starting from a fit to the toy adaptation behavior shown in Fig. 2.3, these curves were produced by adjusting only the model parameters θ and C'_{13}. As these two parameters are increased together, the sensitivity and precision of the output increases (sensitivity and precision are defined in Fig. 1.1).

function, and what parameter combinations are important to the model output. Each process of model reduction yields a simple network with its own minimal mechanism (although different model reductions may converge to the same minimal network). The limits themselves and the equations they imply articulate what is behind the adaptation behavior of a certain network.

Future research will require many more reductions to be completed so that a large representative sampling of minimal networks can be found. This is being worked towards by increasing the number of different types of limits that can be automatically discerned by the automated model reduction program.

3.2 Conclusion

Here we recapitulate what we have accomplished. We have identified a new minimal mechanism that achieves adaptation in the form of the network in Fig. 3.1 with its attendant equations. We have thus demonstrated a practical implementation of model reduction with the model boundary approximation method (MBAM), showing the insight it provides into network behaviors. We have also shown that model reduction from a fully-connected network (FCN), although the FCN by itself is not physically meaningful, leads to physically meaningful mechanisms. The conclusions of this research agree with and expand on other key systems biology research on adaptation.

This process has illustrated that the implementation of MBAM can be automated. In principle, this can be done to search for any behavior of enzyme networks, including oscillations, memory, or other behaviors, simply by changing the toy function to which parameters are fitted, and possibly the input function.

Also, although the code used in this research is intended particularly to identify parameter limits of the Michaelis-Menten dynamic equations, the method can be extended to any system that is modelled by ordinary differential equations, including models in ecology, economics, population

dynamics, and control theory. This research presents an illustration of a general framework that articulates how systems achieve interesting behaviors. It uses the language of parameter limits and emergent equations, which we hope will aid in understanding emergent behaviors across many fields of inquiry.

Appendix A

Limits of the Michaelis-Menten Equations

A number of recurring types of limits beyond those included in Table 2.1 shed light on some of the biophysics behind the adaptive behaviors of three-node networks. We identify three particularly clear types of parameter limits that also qualify as candidates for automation in future work.

First, we have saturation limits, in which all of the reactions feeding into a particular node become very strong. Mathematically, this uniformly overwhelms the rate of change $\frac{dX}{dt}$ and changes the differential equation for that node into an algebraic equation. Physically, this represents reduced reaction times between nodes, which sets up a functional relationship between them.

Second, we have differential limits, in which promotion and inhibition from the same node become strong and balance each other. The emergent parameters from these limits are differences of the Michaelis-Menten parameters.

Finally, we have rescaling limits, in which the concentration of active enzyme in a node becomes very small at the same time the effect of that node on the rest of the network becomes very strong. This can have the effect of bypassing a node, or using a bypassed node to both promote and inhibit another node.

Two examples of each type of limit are included in the following subsections.

A.0.1 Differential Limits

Eq: $$\frac{dX_2}{dt} = -\theta_0 X_2 + \frac{C_{12}X_1(1-X_2)}{-X_2+1+\varepsilon} + \frac{C_{32}X_3(1-X_2)}{K_{32}-X_2+1} - C'_{32}X_3$$

Limit: $K_{32} \to 0$,

$C_{32}, C'_{32} \to \infty$

Combining the second and third to last terms, we take $X_{2I} = 1 - X_2$, and

$$\frac{dX_2}{dt} = -\theta_0 X_2 + \frac{C_{12}X_1(1-X_2)}{-X_2+1+\varepsilon} + \frac{C_{32}X_{2I} - C'_{32}K_{32} - C'_{32}X_{2I}}{K_{32}+X_{2I}}X_3.$$

After evaluating the limit,

$$\boxed{\frac{dX_2}{dt} = -\theta_0 X_2 + \frac{C_{12}X_1(1-X_2)}{-X_2+1+\varepsilon} + \frac{\theta_1 X_{2I} - \theta_2}{X_{2I}+\varepsilon}X_3}$$

where

$\theta_1 = C_{32} - C'_{32}$,

$\theta_2 = C'_{32}K_{32}$.

DIFFERENTIAL LIMIT

Eq: $$\frac{dX_3}{dt} = \theta_0 X_3 + \frac{C_{13}X_1(1-X_3)}{K_{13}-X_3+1} + \frac{C_{23}X_2(1-X_3)}{-X_3+1+\varepsilon} - \frac{C'_{13}X_1X_3}{K'_{13}+X_3} - \frac{C'_{23}X_2X_3}{K'_{23}+X_3}$$

Limit: $K_{13}, K'_{13} \to 0,$

$C_{13}, C'_{13} \to \infty$

Combining the second and fourth terms together with $X_{I3} = 1 - X_3$ yields

$$X_1 \frac{C_{13}X_{I3}K'_{13} + C_{13}X_{I3}X_3 - C'_{13}X_3K_{13} - C'_{13}X_3X_{I3}}{(K_{13}+X_{I3})(K'_{13}+X_3)}.$$

After evaluating the limit,

$$\boxed{\frac{dX_3}{dt} = \theta_0 X_3 + \frac{C_{23}X_2(1-X_3)}{-X_3+1+\varepsilon} - \frac{C'_{23}X_2X_3}{K'_{23}+X_3} + X_1\frac{\theta_1 X_{I3} - \theta_2 X_3 + \theta_3 X_{I3}X_3}{X_{I3}X_3+\varepsilon}}$$

where

$\theta_1 = C_{13}K'_{13},$

$\theta_2 = C'_{13}K_{13},$

$\theta_3 = C_{13} - C'_{13}.$

A.0.2 Saturation Limits

Eq: $\frac{dX_2}{dt} = \frac{C_{12}X_1(1-X_2)}{1-X_2+K_{12}} - \theta_0 X_2$

Limit: $C_{12}, \theta_0 \to \infty$

Dividing all terms by θ_0,

$$\frac{1}{\theta_0}\frac{dX_2}{dt} = \left(\frac{C_{12}}{\theta_0}\right)\frac{X_1(1-X_2)}{1-X_2+K_{12}} - X_2.$$

After evaluating the limit, we have

$$\boxed{0 = \theta_1 \frac{X_1(1-X_2)}{1-X_2+K_{12}} - X_2}$$

The differential equation is now **algebraic**, with

$\theta_1 = C_{12}/\theta_0$.

SATURATION LIMIT

Eq: $\dfrac{dX_3}{dt} = \dfrac{C_{13}X_1(1-X_3)}{K_{13}-X_3+1} - C'_{23}X_2 - \dfrac{C'_{E3}X_3}{K'_{E3}+X_3}$

Limit: $C_{13}, C'_{23}, C'_{E3} \to \infty$

After evaluating the limit,

$$\boxed{0 = \frac{X_1(1-X_3)}{K_{13}-X_3+1} - \theta_1 X_2 - \frac{\theta_2 X_3}{K'_{E3}+X_3}}$$

The differential equation is now **algebraic**, with

$\theta_1 = C'_{23}/C_{13}$,

$\theta_2 = C'_{E3}/C_{13}$.

A.0.3 Rescaling Limits

Eqs:
$$\frac{dX_1}{dt} = \frac{C_{21}X_2(1-X_1)}{-X_1+1+\varepsilon} + \frac{I_{IC}(1-X_1)}{K_{I1}-X_1+1}$$

$$\frac{dX_2}{dt} = \frac{C_{12}X_1(1-X_2)}{-X_2+1+\varepsilon} + \frac{C_{32}X_3(1-X_2)}{-X_2+1+\varepsilon} - \frac{C'_{12}X_1X_2}{K'_{12}+X_2} - \frac{C'_{E2}X_2}{K'_{E2}+X_2}$$

$$\frac{dX_3}{dt} = \frac{C_{13}X_1(1-X_3)}{K_{13}-X_3+1} + \frac{C_{23}X_2(1-X_3)}{K_{23}-X_3+1} - \frac{C'_{13}X_1X_3}{K'_{13}+X_3} - \frac{C'_{E3}X_3}{K'_{E3}+X_3}$$

Limit: $C_{12}, C'_{12}, K'_{12}, C'_{E2}, K'_{E2}, C_{32} \to 0,$

$C_{21}, C_{23} \to \infty$

After evaluating the limit, we have

$$\boxed{\frac{dX_1}{dt} = \frac{\widetilde{X_2}(1-X_1)}{-X_1+1+\varepsilon} + \frac{I_{IC}(1-X_1)}{K_{I1}-X_1+1}}$$

$$\boxed{\frac{d\widetilde{X_2}}{dt} = \theta_1X_1 + \theta_2X_3 - \frac{\theta_3X_1\widetilde{X_2}}{\theta_4+\widetilde{X_2}} - \frac{\theta_5\widetilde{X_2}}{\theta_6+\widetilde{X_2}}}$$

$$\boxed{\frac{dX_3}{dt} = \frac{C_{13}X_1(1-X_3)}{K_{13}-X_3+1} + \frac{\theta_7\widetilde{X_2}(1-X_3)}{K_{23}-X_3+1} - \frac{C'_{13}X_1X_3}{K'_{13}+X_3} - \frac{C'_{E3}X_3}{K'_{E3}+X_3}}$$

where $\widetilde{X_2} = C_{21}X_2$, and

$\theta_1 = C_{12}C_{21},$

$\theta_2 = C_{32}C_{21},$

$\theta_3 = C'_{12}C_{21},$

$\theta_4 = K'_{12}C_{21},$

$\theta_5 = C'_{E2}C_{21},$

$\theta_6 = K'_{E2}C_{21},$

$\theta_7 = C_{23}/C_{21}.$

RESCALING LIMIT

Eqs:
$$\frac{dX_1}{dt} = \frac{C_{31}X_3(1-X_1)}{-X_1+1+\varepsilon} - \frac{C'_{21}X_1X_2}{K'_{21}+X_1} + \frac{I_{IC}(1-X_1)}{K_{I1}-X_1+1}$$

$$\frac{dX_2}{dt} = \frac{C_{12}X_1(1-X_2)}{-X_2+1+\varepsilon} + \frac{C_{32}X_3(1-X_2)}{K_{32}-X_2+1} - \frac{C'_{E2}X_2}{K'_{E2}+X_2}$$

$$\frac{dX_3}{dt} = \frac{C_{13}X_1(1-X_3)}{-X_3+1+\varepsilon} + \frac{C_{23}X_2(1-X_3)}{K_{23}-X_3+1} - \frac{C'_{23}X_2X_3}{K'_{23}+X_3}$$

Limit: $C'_{21}, K'_{21}, C_{31} \to 0,$

$C_{12}, C_{13}, K_{I1} \to \infty$

After evaluating the limit, we have

$$\boxed{\frac{d\widetilde{X_1}}{dt} = \theta_1 X_3 - \frac{\theta_2 \widetilde{X_1} X_2}{\theta_3 + \widetilde{X_1}} + \frac{I_{IC}}{\theta_4}}$$

$$\boxed{\frac{dX_2}{dt} = \frac{\widetilde{X_1}(1-X_2)}{-X_2+1+\varepsilon} + \frac{C_{32}X_3(1-X_2)}{K_{32}-X_2+1} - \frac{C'_{E2}X_2}{K'_{E2}+X_2}}$$

$$\boxed{\frac{dX_3}{dt} = \frac{\theta_5 \widetilde{X_1}(1-X_3)}{-X_3+1+\varepsilon} + \frac{C_{23}X_2(1-X_3)}{K_{23}-X_3+1} - \frac{C'_{23}X_2X_3}{K'_{23}+X_3}}$$

where $\widetilde{X_1} = C_{12}X_1$, and

$\theta_1 = C_{31}C_{12},$

$\theta_2 = C'_{21}C_{12},$

$\theta_3 = K'_{21}C_{12},$

$\theta_4 = K_{I1}/C_{12},$

$\theta_5 = C_{13}/C_{12}.$

Appendix B

Code

This appendix contains original Python code that runs automated model reduction. The general structure of the code has `Reduce.py` initiating and overseeing reductions, with the primary work being performed by the various functions in `MMAutoReduce.py`. Other important files serving incidental functions are also included. A flowchart for the more detailed structure of the code is included in Fig 2.8.

This code relies on the open-source NumPy, SciPy, and `geodesiclm` packages, as well as the `symbolic` package written by Dane Bjork, and the following code written by Mark K. Transtrum: class definitions `BaseModel`, `Composition`, `modeling`, `geometry`, and `DAECalc`.

Reduce.py

```
import numpy as np
import sys
sys.path.append('/data/merrill/DAEAutoReduction')
import MMAutoReduce
reload(MMAutoReduce)
R = MMAutoReduce.MMAutoReduce
from EnzX_Model import BuildModel


fast = True
sval = False
debug = False
bkw = False


N = int(sys.argv[1])
if 'slow' in sys.argv:
    fast = False
if 'svp' in sys.argv:
    sval = True
if 'debug' in sys.argv:
    debug = True
if 'b' in sys.argv:
    bkw = True


def assemblemodel(N):
    return BuildModel('Enz3_%02i' % N, N, 3, np.linspace(0,25,250))
```

```
reduction = R(assemblemodel,'Enz3',N,4,path='../')


if bkw:
    x = np.load('../xValues/xtrue_%02i.npy' % N)
    try:
        reduction.iterate(x,fast,'b',1,sval)
        N -= 1
    except:
        raise


for i in range(N-2):
    x = np.load('../xValues/xtrue_%02i.npy' % (N-i))
    try:
        reduction.iterate(x,fast,'s',1,sval)
    except:
        if not debug:
            try:
                reduction.iterate(x,fast,'b',1,sval)
            except:
                try:
                    reduction.iterate(x,fast,'s',3,sval)
                except:
                    try:
                        reduction.iterate(x,fast,'b',3,sval)
                    except:
                        reduction.requestManual()
                        raise
```

```
        else:
            raise

    fast = True
    sval = False
```

MMAutoReduce.py

```
import numpy as np
import logging
from symbolic import Sym
from modeling.DAECalc import MakeandCompile
from geodesiclm import geodesiclm
from datetime import datetime
from TransTemplate import template

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

from scipy.optimize import brentq

class NoEdgeFound(Exception):
    def __init__(self):
        pass

class EdgeNotRecognized(Exception):
    def __init__(self):
        pass
```

```
def vdota(v, j, A):
    g = np.dot(j.T,j)
    a = -np.linalg.solve(g, np.dot(j.T, np.dot(np.dot(A, v), v)))
    return np.dot(v, a)

def vPerp(v1,v2):
    v2new = v2 - v1*np.dot(v1,v2)/(np.linalg.norm(v1))**2
    v2new = v2new/np.linalg.norm(v2new)
    return v2new

def FindPerp(lst):  #creates unit vector perpendicular to all in lst
    N = len(lst[0])
    vtest = np.random.randn(N)
    vtest = vtest/np.linalg.norm(vtest)
    if(np.linalg.norm(lst[0]) != 0):
        for v in lst:
            vperp = vtest - v*np.dot(vtest, v)/np.linalg.norm(v)**2
            vperp = vperp/np.linalg.norm(vperp)
            vtest = vperp
    return vtest

def vdath(th,vp,vp2,j,A): #returns interpolation between ortho vectors vp, vp2
    v = np.cos(th)*vp + np.sin(th)*vp2
    return vdota(v,j,A)

def Search(lst,j,A):
```

```
global spv
vp = FindPerp(lst)            #vector perp to all vectors in lst
vp = vp/np.linalg.norm(vp)
vp2 = FindPerp(lst)           #another
vp2 = vPerp(vp,vp2)             #made perp with vp
vp2 = vp2/np.linalg.norm(vp2)
vs = []
i=0
sg = np.sign(vdota(vp,j,A))
a = 0
th = 0.1
while(th < np.pi):  #fills vs with all vectors v for which v.a=0
                    #in the plane of vp, vp2
    vda = vdath(th,vp,vp2,j,A)
    if(np.sign(vda) == sg):
        a = th
    else:
        b = th
        th0 = brentq(vdath,a,b,args=(vp,vp2,j,A))
        #remove vectors that are similar in dspace
        vtest=np.cos(th0)*vp + np.sin(th0)*vp2
        addOn=1
        for v in vs:
            if(np.abs(np.dot(np.dot(j,vtest),np.dot(j,v)))>0.95):
                addOn=0
        if(addOn==1):
            vs.append(vtest)
```

```
            #
            i += 1
            a = th
        th += 0.1
        sg = np.sign(vda)
    return vs


def Branch(j,A):
    global nvec
    global vdas
    N = j.shape[1]
    if(N==2):
        vs = Search([np.zeros(2)],j,A)
        while(len(vs)==0):
            vs = Search([np.zeros(2)],j,A)
        bases = []
        for v in vs:
            bases.append([v])
    else:
        vp = np.random.randn(N)
        vp = vp/np.linalg.norm(vp)
        chk = Search([vp],j,A)
        if(chk == []):
            chk = Search([vp],j,A)
        bases = [[chk[0]]]
        for i in range(N-2):
            addbases = []
```

```
            for lst in bases:
                poss = Search(lst,j,A)
                while(poss==[] or np.isnan(poss[0][0])):
                    poss = Search(lst,j,A)
                if(len(poss)>1):
                    for v in poss[1:]:
                        addbases.append(lst + [v])
                lst.append(poss[0])
            bases = bases + addbases
    return bases


def Findv0s(j,A):
    bases = Branch(j,A)
    v0s = []
    for basis in bases:
        m = np.zeros((j.shape[1]-1, j.shape[0]))
        for k in range(j.shape[1]-1):
            m[k] = np.dot(j,basis[k])
        m = m.T
        u,s,vh = np.linalg.svd(m,0)
        vrand = np.random.randn(j.shape[1])
        d = np.dot(j,vrand)
        dp = np.dot(np.eye(u.shape[0]) - np.dot(u,u.T), d)
        uj,sj,vhj = np.linalg.svd(j,0)
        v0 = np.zeros(j.shape[1])
        for n in range(j.shape[1]):
            v0 += np.dot(dp,uj.T[n])/sj[n]*vhj[n]
```

```
        v0 = v0/np.linalg.norm(v0)
        if(vdota(v0,j,A)<0):
            v0 *= -1
        v0s.append(v0)
    return v0s


def Discern(x,v,model):
    Disc = {}
    j = model.j(x)
    u,s,vh = np.linalg.svd(j)
    vnorm = vh[-1]/np.linalg.norm(vh[-1])
    if np.dot(vnorm,v)<0:
        vnorm *= -1


    n=-1
    arglist = list(np.argsort(np.abs(vnorm)))
    arglist.reverse()
    for i in range(len(vnorm)-1):
        if np.abs(vnorm[arglist[i]]) - np.abs(vnorm[arglist[i+1]]) > 0.8/np.sqrt(i+1):
            n = i+1
            break
    if n==-1:
        return Disc


    for j in arglist[:n]:
        if vnorm[j]>0:
            Disc[j] = np.inf
```

```
        else:
            Disc[j] = 0


    return Disc


def JDiscern(xi,xf,v,model):
    Disc = {}
    ji = model.j(xi)
    jf = model.j(xf)
    u,s,vh = np.linalg.svd(jf-ji)
    vnorm = vh[-1]/np.linalg.norm(vh[-1])
    if np.dot(vnorm,v)<0:
        vnorm *= -1


    n=-1
    arglist = list(np.argsort(np.abs(vnorm)))
    arglist.reverse()
    for i in range(len(vnorm)-1):
        if np.abs(vnorm[arglist[i]]) - np.abs(vnorm[arglist[i+1]]) > 0.8/np.sqrt(i+1):
            n = i+1
            break
    if n==-1:
        return Disc


    for j in arglist[:n]:
        if vnorm[j]>0:
            Disc[j] = np.inf
```

```
        else:
            Disc[j] = 0

    return Disc

def SDiscern(vec):
    Disc = {}
    vnorm = vec/np.linalg.norm(vec)

    n=-1
    arglist = list(np.argsort(np.abs(vnorm)))
    arglist.reverse()
    for i in range(len(vnorm)-1):
        if np.abs(vnorm[arglist[i]]) - np.abs(vnorm[arglist[i+1]]) > 0.8/np.sqrt(i+1):
            n = i+1
            break
    if n==-1:
        return Disc

    for j in arglist[:n]:
        if vnorm[j]>0:
            Disc[j] = np.inf
        else:
            Disc[j] = 0

    return Disc
```

```
class MMAutoReduce:


    def __init__(self,assemblemodel,modelname,N,Noffset=0,lim=25,path='../',rand=False):
        '''
        modelname is the base name of the models,
        then _xx is appended, where xx is the number of parameters.

        assemblemodel is a function that produces a model given
        the number of parameters
        '''
        self.lim = lim
        self.Noffset = Noffset
        self.modelname = modelname
        self.assemblemodel = assemblemodel
        self.loadmodel(modelname + '_%02i' % N)
        self.path = path
        if(self.path[-1]!='/'):
            self.path += '/'
        if rand:
            randLogString = '_boundaries'
        else:
            randLogString = ''
        handler = logging.FileHandler(path + modelname + randLogString + '.log')
        handler.setLevel(logging.DEBUG)
        logger.addHandler(handler)
        logger.info('\n*** ' + str(datetime.now()) + ' ***')
```

```
def loadmodel(self,fullmodelname):
    self.modeldef = __import__(fullmodelname)
    self.N = len(self.modeldef.parameters) - self.Noffset
    self.model = self.assemblemodel(self.N)


def v0(self,method='s'):
    self.v0method = method
    if(method=='s' or method=='b'):  #smallest sval direction
        u,s,vh = np.linalg.svd(self.j)
        v = vh[-1]
        jt = np.transpose(self.j)
        jtjin = np.linalg.inv(np.dot(jt,self.j))
        a = -np.dot(jtjin,np.dot(jt,self.model.Avv(self.x,v)))
        if np.dot(a,v) < 0:
            v = v * -1
        if(method=='b'):
            logger.info('Used reverse smallest sval direction for v0.')
            v = v * -1
    if(method=='h'):  #hyperplane norm method
        logger.info('Used hyperplane norm method for v0.')
        A = self.model.A(self.x)
        v = Findv0s(self.j,A)[0]
    if(method=='rand'):
        v = np.random.rand(self.N)
        v[-1] = 1
    return v
```

```
def callback(self,geo):
      lim = self.lim
      Dmethod = self.Dmethod
      if(self.v0method != 's'):
          geoappend = '.' + self.v0method
      else:
          geoappend = ''
      if(Dmethod==3):
          geoappend += 'j'
          if(geoappend[0] != '.'):
              geoappend = '.' + geoappend
      if self.v0method == 'rand':
          geoappend += '%03i' % self.iter


      print geo.t, np.linalg.norm(geo.vs[-1])
      np.save(self.path + "Geodesics/ts%s.%02i"% (geoappend, self.N), geo.ts)
      np.save(self.path + "Geodesics/xs%s.%02i"% (geoappend, self.N), geo.xs)
      np.save(self.path + "Geodesics/vs%s.%02i"% (geoappend, self.N), geo.vs)
      if Dmethod==1:
          Disc = Discern(geo.xs[-1], geo.vs[-1], self.model)
      elif Dmethod==2:
          Disc = SDiscern(geo.vs[-1] - geo.vs[-2])
      else:
          Disc = JDiscern(geo.xs[-2], geo.xs[-1], geo.vs[-1], self.model)
      if(self.isfast):
          return (Disc=={} and np.linalg.norm(geo.vs[-1]) < lim*self.v0norm)
      else:
```

```
            return np.linalg.norm(geo.vs[-1]) < lim*self.v0norm


    def calcgeodesic(self):
        if(self.svalpath):
            logger.info('Follows direction corresponding to the smallest singular value of j')
            from svalpath import geodesic
        else:
            from geometry import geodesic
        model = self.model
        x=self.x
        v=self.v
        geo = geodesic(model.r,model.j,model.Avv,model.M,model.N,x,v,atol=1e-3,rtol=1e-3,
            callback = self.callback)
        geo.invSVD = True
        geo.integrate(100, 400)
        logger.info('Residual norm geodesic end: %g' % np.linalg.norm(self.model.r(geo.xs[-1])))
        return geo


    def saveandcompile(self,sym,reparam=True):
        N = self.N
        if(self.rename==''):
            self.rename = 'Enz3_%02i' % (N-1)
        if reparam:
            for i in range(N-1):
                #This dummy renaming ensures no parameter is renamed twice
                sym.replaceparam(i+self.Noffset, "P%d"%i)
            for i in range(N-1):
```

```
            #Renames with correct names
            sym.replaceparam(i+self.Noffset, "p%d"%i)
    sym.savetofile(self.rename,True)
    MakeandCompile(self.rename,True,True)

def Fit(self):
    model = self.model
    x = np.load(self.path + 'xValues/xinit_%02i.npy' % self.N)
    logger.info('Residual norm pre-fit: %g' % np.linalg.norm(self.model.r(x)))

    try:
        xf,info = geodesiclm(model.r,x,jacobian = model.j,Avv = model.Avv, full_output = 1,
            print_level = 0, maxiter = 300, ibold = 0)
        xfs = xf
    except:
        logger.error('Fitting failed')
        raise

    logger.info('Residual norm post-fit: %g' % np.linalg.norm(self.model.r(xfs)))
    np.save(self.path + 'xValues/xtrue_%02i' % self.N, xfs)

def evallim(self):
    Disc = self.Disc
    sym = Sym(self.modeldef)
    logger.debug('Discern: %s' % Disc)
    if Disc == {}:
        logger.error('No edge was discerned.')
```

```
        raise NoEdgeFound

    #if one parameter goes to zero
    if len(Disc.keys())==1:
        if Disc.values()[0] != 0:
            raise EdgeNotRecognized
        idx = Disc.keys()[0] + self.Noffset
        pname = sym.parameters[idx]
        logger.info('%s -> 0' % pname)
        sym.substituteall(sym.parray[idx], 0.0001)
        logger.debug('1e-04 substituted for %s' % sym.parameters[idx])
        logger.debug('%s removed' % sym.parameters[idx])
        sym.remparam(sym.parameters[idx])

        x = self.geo.xs[-1]
        newx = np.empty(len(x) - 1)
        tstring = ''
        for i in range(len(newx)):
            if(i < Disc.keys()[0]):
                newx[i] = x[i]
                tstring += '            x[%d],%s' % (i,'\n')
            else:
                newx[i] = x[i+1]
                tstring += '            x[%d],%s' % (i+1,'\n')
        np.save(self.path + 'xValues/xinit_%02i.npy' % len(newx), newx)

        temp = template % {'name1':'%02i' % len(x), 'name2':'%02i' % len(newx),
```

```
            'M':len(newx), 'N':len(x), 'transforms':tstring}
        fo = open("../Transforms/Trans%02i_to_%02i.py" % (len(x), len(newx)),'w')
        fo.write(temp)
        fo.close()

    else:
        #if we have a CK pair
        if len(Disc.keys()) != 2:
            raise EdgeNotRecognized
        if(Disc.values()[0] != np.inf or Disc.values()[1] != np.inf):
            raise EdgeNotRecognized
        idx1 = min(Disc.keys()) + self.Noffset
        idx2 = max(Disc.keys()) + self.Noffset
        if not sym.check_ck(idx1,idx2):
            raise EdgeNotRecognized
        logger.info('%s / %s' % (sym.parameters[idx1], sym.parameters[idx2]))
        pnew = 'CK'
        pnew += sym.parameters[idx1][1:]
        sym.ckratio(pnew, idx1, idx2)
        p1 = sym.parameters[idx1]
        p2 = sym.parameters[idx2]
        logger.debug('%s removed' % p1)
        sym.remparam(p1)
        logger.debug('%s removed' % p2)
        sym.remparam(p2)

        x = self.geo.xs[-1]
```

```
            newx = np.empty(len(x) - 1)
            skip = 0
            tstring = ''
            for i in range(len(newx) - 1):
                while(Disc.has_key(i+skip)):
                    skip += 1
                newx[i] = x[i+skip]
                tstring += '             x[%d],%s' % (i+skip,'\n')
            newx[-1] = x[min(Disc.keys())] - x[max(Disc.keys())]
            tstring += '             x[%d] - x[%d]' % (min(Disc.keys()), max(Disc.keys()))
            np.save(self.path + 'xValues/xinit_%02i.npy' % len(newx), newx)

            temp = template % {'name1':'%02i' % len(x), 'name2':'%02i' % len(newx),
                'M':len(newx), 'N':len(x), 'transforms':tstring}
            fo = open("../Transforms/Trans%02i_to_%02i.py" % (len(x), len(newx)),'w')
            fo.write(temp)
            fo.close()

        self.saveandcompile(sym)

    def requestManual(self, TeX=True):
        logger.info('Automatic reduction failed.')
        logger.info('Manual reduction required.')
        if TeX:
            import re
            import os
            from toTeX import toTeX
```

```
        cdir = os.path.realpath('..')
        location = re.findall(r'R.+$', cdir)[-1]
        toTeX(self.modeldef, location, 'Enz3_%02i' % self.N, self.Disc1,
            TeXname='%s_%02i.tex' % (location, self.N))

def iterate(self,x,isfast=True,v0method='s',Dmethod=1,svalpath=False,rename='',iter=0):
    logger.info('Iterating from %i parameters' % self.N)
    self.x = x
    self.j = self.model.j(x)
    self.N = len(x)
    self.isfast = isfast
    if not isfast:
        logger.info('Running geodesic until velocity increases by a factor of %d' % self.lim
    self.Dmethod = Dmethod
    self.svalpath = svalpath
    self.rename = rename
    self.iter = iter

    self.v = self.v0(v0method)
    self.v0norm = np.linalg.norm(self.v)

    self.geo = self.calcgeodesic()

    if Dmethod == 1:
        Disc = Discern(self.geo.xs[-1],self.geo.vs[-1],self.model)
    elif Dmethod == 2:
        logger.info('Using parameter acceleration to discern limit')
```

```
        Disc = SDiscern(self.geo.vs[-1] - self.geo.vs[-2])
    else:
        logger.info('Using sval decomp of jacobian difference to discern limit')
        Disc = JDiscern(self.geo.xs[-2],self.geo.xs[-1],self.geo.vs[-1],self.model)
    self.Disc = Disc

    #Saves the Discern dictionary of the first geodesic, which tends to be the best for
    #finding manual limits.
    if v0method == 's':
        self.Disc1 = Disc

    self.evallim()

    self.loadmodel(self.modelname + '_%02i' % (self.N - 1))
    self.Fit()
```

Export.py

```
import numpy as np
import shutil
import subprocess
import sys

fam = int(sys.argv[1])
fits = np.loadtxt('xf%04i.txt' % fam)
rng = range(int(sys.argv[2]),int(sys.argv[3]))

for i in rng:
    if(not np.isnan(fits[i][0])):
        nm = 'R.%d.%03i' % (fam,i)
        shutil.copytree('NewReduction', '../../%s' % nm)
        np.save('../../%s/xValues/xtrue_31.npy' % nm, fits[i])
        shutil.os.chdir('../../%s/ModelDefs' % nm)
        subprocess.Popen(['nohup', 'python', 'Reduce.py', '31'])
        shutil.os.chdir('../../Enz3/FitParameters')
```

Enz3_31.py

```
parameters = [
    "X1_IC",
    "X2_IC",
    "X3_IC",
    "I_IC",
    "Cp_E_1",
    "Kp_E_1",
    "C_1_2",
    "K_1_2",
    "Cp_1_2",
    "Kp_1_2",
    "C_1_3",
    "K_1_3",
    "Cp_1_3",
    "Kp_1_3",
    "C_2_1",
    "K_2_1",
    "Cp_2_1",
    "Kp_2_1",
    "Cp_E_2",
    "Kp_E_2",
    "C_2_3",
    "K_2_3",
    "Cp_2_3",
    "Kp_2_3",
    "C_3_1",
```

```
    "K_3_1",
    "Cp_3_1",
    "Kp_3_1",
    "C_3_2",
    "K_3_2",
    "Cp_3_2",
    "Kp_3_2",
    "Cp_E_3",
    "Kp_E_3",
    "K_I_1",
    ]

assignments = []

DVars = [
    ("X1", 1, "X1_IC"),
    ("X2", 1, "X2_IC"),
    ("X3", 1, "X3_IC"),
    ]

RESs = [
    "I_IC*(1-X1)/((1-X1)+K_I_1) + X2*C_2_1*(1-X1)/((1-X1)+K_2_1) +
X3*C_3_1*(1-X1)/((1-X1)+K_3_1) - X2*Cp_2_1*X1/(X1+Kp_2_1) - X3*Cp_3_1*X1/(X1+Kp_3_1)
- Cp_E_1*X1/(X1+Kp_E_1) - X1_prime",
    "X1*C_1_2*(1-X2)/((1-X2)+K_1_2) + X3*C_3_2*(1-X2)/((1-X2)+K_3_2) - X1*Cp_1_2*X2/(X2+Kp_1_2)
- X3*Cp_3_2*X2/(X2+Kp_3_2) - Cp_E_2*X2/(X2+Kp_E_2) - X2_prime",
    "X1*C_1_3*(1-X3)/((1-X3)+K_1_3) + X2*C_2_3*(1-X3)/((1-X3)+K_2_3) - X1*Cp_1_3*X3/(X3+Kp_1_3)
```

```
- X2*Cp_2_3*X3/(X3+Kp_2_3) - Cp_E_3*X3/(X3+Kp_E_3) - X3_prime",
    ]


if __name__ == "__main__":
    from modeling.DAECalc import MakeandCompile
    MakeandCompile("Enz3_31", True, True)
```

EnzX_Model.py

```
import numpy as np
from modeling import Composition
from modeling.BaseModel import BaseModel
from modeling.DAECalc import DAECalc


class EnzX(BaseModel):
    def __init__(self,name,rel,nodes,ts):
        self.name = name
        self.rel = rel
        self.nodes = nodes
        self.ts = ts
        self.calc = DAECalc("__%s__" % (name), "__d%s__" % (name), "__d2%s__" % (name))
        self.calc.kwargs['max_steps'] = 10000
        BaseModel.__init__(self, len(ts)*nodes, rel+nodes+1, "Arbitrary enzyme reaction network"


    def r(self, x):
        sol = self.calc.evaluate(x, self.ts)
        return sol.flatten()
```

```
    def v(self, x, v):
        solv = self.calc.evaluate_derivative(x, v, self.ts)
        return solv.flatten()

    def Avv(self, x, v):
        solAvv = self.calc.evaluate_Avv(x, v, self.ts)
        return solAvv.flatten()

    def Auv(self, x, u, v):
        solAuv = self.calc.evaluate_Auv(x, u, v, self.ts)
        return solAuv.flatten()

def BuildModel(name,rel,nodes,ts):
    baremodel = EnzX(name,rel,nodes,ts)
    from Exp_Model import Exponential
    from SetICX import SetICX
    from ResidualsX import ResidualsX

    prep = Composition(SetICX(baremodel.rel, nodes), Exponential(baremodel.rel))
    model = Composition(baremodel, prep)
    residuals = Composition(ResidualsX(nodes,baremodel.ts), model)

    return residuals
```

SetICX.py

```
import numpy as np
from modeling.BaseModel import BaseModel


class SetICX(BaseModel):
    def __init__(self, N, nodes):
        self.nodes = nodes
        BaseModel.__init__(self, N + nodes + 1, N, "Adds initial conditions to the input vector"


    def r(self, x):
        IC = np.concatenate((np.zeros(self.nodes),np.array([0.5])))
        return np.concatenate([IC, x])


    def v(self, x, v):
        return np.concatenate([np.zeros((self.nodes+1)),v])


    def Avv(self, x, v):
        return np.zeros((self.M))


    def Auv(self, x, u, v):
        return np.zeros((self.M))
```

ResidualsX.py

```
import numpy as np
from modeling.BaseModel import BaseModel


def dfault(t):
    return 0.5*np.exp(-0.2*(t-5)**2) + 0.25*(1-np.exp(-t))**5
def rsine(t):
    return -0.5*np.exp(-t) + 0.1*np.sin(t*2*np.pi/10) + 0.5


class ResidualsX(BaseModel):
    def __init__(self, nodes, ts, s=0.1, dat=dfault):
        self.nodes = nodes
        self.ts = ts
        if s == 0:
            s = 0.25
        self.s = np.zeros((len(self.ts))) + s
        self.dat = dat
        BaseModel.__init__(self, len(self.ts), len(self.ts)*nodes,
            "Subtracts data from final section of input vector and divides by sigmas")


    def r(self, x):
        return (x[(self.nodes-1)::self.nodes] - self.dat(self.ts))/self.s


    def v(self, x, v):
        return v[(self.nodes-1)::self.nodes]/self.s


    def Avv(self, x, v):
```

```
    return self.s*0


def Auv(self, x, u, v):
    return self.s*0
```

Additional function added to **symbolic.py**

```
def check_ck(self, idx1, idx2):
    '''
    returns a boolean = True if param[idx1] and param[idx2] occur
    only in the same residual terms, False otherwise.
    '''
    for res in range(len(self.rarray)):
        spec1 = [] #indices of terms that contain param[idx1]
        spec2 = [] #indices of terms that contain param[idx2]

        for j in range(len(self.rarray[res].args)):
            if(self.rarray[res].args[j].has(self.parray[idx1])):
                spec1.append(j)
            if(self.rarray[res].args[j].has(self.parray[idx2])):
                spec2.append(j)

        if len(spec1) == len(spec2):
            tested = True
            for k in range(len(spec1)):
                if spec1[k] != spec2[k]:
                    tested = False
        else:
```

```
                tested = False


            return tested
```

TransTemplate.py

```
template="""
import numpy as np
from modeling import BaseModel

class Trans%(name1)s_to_%(name2)s(BaseModel):

    def __init__(self):
        BaseModel.__init__(self,%(M)i,%(N)i,'%(name1)s_to_%(name2)s')

    def r(self,x):
        return [
%(transforms)s
        ]

trans = Trans%(name1)s_to_%(name2)s()
"""
```

TeX_Template.py

```
template="""
\documentclass[12pt]{article}
\usepackage{amsfonts,amssymb,amsthm,amsmath}

\setlength{\oddsidemargin}{-0.1in} \setlength{\\textwidth}{6.5in}
\setlength{\\topmargin}{-.75in} \setlength{\\textheight}{9.75in}

\\newenvironment{problems}{\\begin{list}{}{\setlength{\labelwidth}{.7in}}}{\end{list}}
\\newcounter{problempart}
\\newenvironment{parts}{\\begin{list}{(\\alph{problempart})}{\setlength{\itemsep}{0pt}
    \usecounter{problempart}}}{\end{list}}
\linespread{1.1}
\\newcommand{\\notR}{\\not{\hskip -3pt R}}

\\begin{document}

\\noindent {
\sc %(location)s   \hfill
RESs%(slice)s            \hfill
%(filename)s
}

\\bigskip

\\begin{problems}
```

```
\item [ RES: ]
%(res)s
\item[ Limit: ]

%(lim0)s
%(lim_inf)s

\\bigskip

After evaluating the limit,

\\bigskip

\\framebox[1.1\width]{Final limit here}

\\bigskip

where

\\bigskip

$\\theta_1 = ,$ \\\\
$\\theta_2 = $.

\end{problems}
\end{document}
"""
```

Appendix C

Network Definitions

The explicit coupled ordinary differential equations for the three-node fully connected enzyme network are written here. The dynamic variables X_i record the fraction of active enzyme in each node. The variable X_1 is the concentration of the input node, excited with the constant stimulus I_{IC} (set to a value of 0.5). The variable X_3 corresponds to the output node, which is checked for adaptive behavior, with the variable X_2 corresponding to the intermediary node. All other variables are tunable model parameters, which can assume any non-negative value. In particular, the C'_{Ei} represent constant inhibition reactions from the environment, and the C and K variables are the normal Michaelis-Menten rate constants for node-to-node reactions.

$$\frac{dX_1}{dt} = I_{IC}\frac{1-X_1}{1-X_1+K_{IC}} + C_{21}\frac{X_2(1-X_1)}{1-X_1+K_{21}} + C_{31}\frac{X_3(1-X_1)}{1-X_1+K_{31}} - C'_{21}\frac{X_2X_1}{X_1+K'_{21}} - C'_{31}\frac{X_3X_1}{X_1+K'_{31}} - C'_{E1}\frac{X_1}{X_1+K'_{E1}}$$

$$\frac{dX_2}{dt} = C_{12}\frac{X_1(1-X_2)}{1-X_2+K_{12}} + C_{32}\frac{X_3(1-X_2)}{1-X_2+K_{32}} - C'_{12}\frac{X_1X_2}{X_2+K'_{12}} - C'_{32}\frac{X_3X_2}{X_2+K'_{32}} - C'_{E2}\frac{X_2}{X_2+K'_{E2}}$$

$$\frac{dX_3}{dt} = C_{13}\frac{X_1(1-X_3)}{1-X_3+K_{13}} + C_{23}\frac{X_2(1-X_3)}{1-X_3+K_{23}} - C'_{13}\frac{X_1X_3}{X_3+K'_{13}} - C'_{23}\frac{X_2X_3}{X_3+K'_{23}} - C'_{E3}\frac{X_3}{X_3+K'_{E3}}$$

Bibliography

Alon, U. 2006, An Introduction to Systems Biology: Design Principles of Biological Circuits (CRC Press)

Artyukhin, A. B., Wu, L. F., & Altschuler, S. J. 2009, Cell, 138, 619

Ma, W., Trusina, A., El-Samad, H., Lim, W. A., & Tang, C. 2009, Cell, 138, 760

Marquardt, D. W. 1963, Journal of the Society for Industrial and Applied Mathematics, 11, 431

Transtrum, M. K., Machta, B., Brown, K., Daniels, B. C., Myers, C. R., & Sethna, J. P. 2015, J. Chem. Phys., 143

Transtrum, M. K., & Qiu, P. 2014, Phys. Rev. Lett., 113

Index