Computational Parameter Selection and Simulation of Complex Sphingolipid Pathway Metabolism

A Thesis
Presented to
The Academic Faculty

by

Peter A. Henning

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Biomedical Engineering

Department of Biomedical Engineering
Georgia Institute of Technology
August 2006

Computational Parameter Selection and Simulation of Complex Sphingolipid Pathway Metabolism

Approved by:

Professor Larry V. McIntire, Advisor
School of Biomedical Engineering
Emory University and Georgia Institute of Technology

Professor Stephen P. DeWeerth
School of Biomedical Engineering
Emory University and Georgia Institute of Technology

Professor Eberhard O. Voit
School of Biomedical Engineering
Emory University and Georgia Institute of Technology

Date Approved : May 17, 2006

To my family,

Dale, Christine, and Sara,

I couldn't have made it through these trying times without you.

ACKNOWLEDGEMENTS

I would like to acknowledge Gautam Goel and Eberhard Voit for their useful insights into past, present, and future of Systems Biology. I would like to thank Dr. Al Merrill and his group for not only providing the sphingolipid experimental data but also for providing expertise in both sphingolipids and mass spectometry. I'd like to thank my fellow researchers Jin Young Hong, John Phan, Pushkar Mukewar, David Stiles, and Paul Tan for answering my at times ridiculous questions on software and various other topics. I would like to also thank my fellow BME classmates for making the semesters fulfilling requirements not only tolerable but at times fun. I'd like to extend my gratitude to Bob Lee and Maggie Cam for their insight into what changes were necessary to make myself a successful researcher and inspiring me to take possession of my graduate degree. I would like to extend my thanks to the BME Department, Georgia Tech, Emory, and NIDDK for this opportunity to be part of cutting edge research. Last but certainly not least, I am grateful for the support from all of my friends.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Systems biology is an emerging field of study that seeks to provide systems-level understanding of biological systems through the integration of high-throughput biological data into predictive computational models. The integrative nature of this field is in sharp contrast as compared to the Reductionist methods that have been employed since the advent of molecular biology. Systems biology investigates not only the individual components of the biological system, such as metabolic pathways, organelles, and signaling cascades, but also considers the relationships and interactions between the components in the hope that an understandable model of the entire system can eventually be developed. This field of study is being hailed by experts as a potential vital technology in revolutionizing the pharmaceutical development process in the post-genomic era.

This work not only provides a systems biology investigation into principles governing de novo sphingolipid metabolism but also the various computational obstacles that are present in converting high-throughput data into an insightful model. First, the work lays out the basics of the remaining work in a comprehensive introduction. The components of this introduction include the following: systems biology's previous studies as well as the potential the future of field holds are explored, the basics of simple chemical reactions as well as established enzymatic reaction models are presented, a brief survey into the elementary concepts of sphingolipids is provided, and basics of the optimization methods employed are discussed. Secondly, a computational framework is set up to accommodate the parameter selection and simulation of a model created from experimental results. The third chapter investigates the validation of the framework with simulated data. Chapter four compares numerical integration schemes on various test problems to determine which numerical integration method is most ideal in solving systems biology problems. The fifth chapter examines the potential use of high performance computing techniques to enhance the previous efforts. Finally, all of the previously mentioned techniques are combined to

prepare a case study that investigates a biochemical model, which accurately models both the normal state of sphingolipid synthesis in human embryonic kidney cells and the abnormal state in cells created by over-expressing a necessary enzyme, serine palmitoyl transferase.

CHAPTER I

INTRODUCTION

The introductory chapter discusses four major topics. First, the motivations and objectives of systems biology are discussed. This section includes a short history of how and why systems biology came into existence as well as a glance at previous work in the field. The second section investigates the basic principles of sphingolipids and their de novo synthesis pathways, which are used as a proof of concept biological system throughout the work. The third section examines common mathematical models used in modeling the enzymatic reactions that are commonplace in systems biology. The fourth section introduces various optimization schemes that are used throughout the remaining chapters.

1.1 Systems Biology Background and Previous Work

1.1.1 Why the time for Systems Biology is Now?

As genetics has progressed from Mendel's time until today, more evidence suggests that complex networks rather than single gene expression are responsible for determining phenotype [7]. Not only have large-scale biological network simulations been created but they have also shown non-intuitive biological insights in areas such as bifurcation analysis of the cell cycle and metabolic analysis [9, 65, 66]. When these insights have been experimentally examined, the simulations were consistent with experimental data [21].

Recent advances in genetics, most notably the completion of the human genome project, have fueled an interest in genetic disposition as the root of many diseases. Several of biotechnology's latest high throughput procedures, such as DNA microarrays, have made through genetic screening a more realistic opportunity. Microarrays are advantageous because they dramatically increase the amount of information obtained in a relatively short time span [79].

Tremendous advances in molecular biology, both in understanding and development of

high throughput data acquisition methods, provide ample means for the studying complex biological control networks, which have been a curiosity for some time [27, 46] . Many recent studies suggest that complex interaction networks are the key component in connecting cellular gene expression to observed phenotype [17, 22, 20]. However, a static interaction map may not provide enough information. A control network's static molecular interaction map is the equivalent of looking at the blueprints of a machine whereas a dynamic pathway model is the equivalent of dismantling the machine and analyzing it piece by piece. One can achieve some understanding by looking only at the blueprints but nothing compared to what is possible with through testing the individual components [46]. The dynamic pathway model, the core of systems biology, can explain the relationship between structure, function, and regulation in complex cellular networks by combining experimental and theoretical approaches [46].

Traditional scientific pursuits have typically involved a cycle with three distinct stages, observation, theory, and physical experimentation. The process starts as an observation, which leads to a hypothesis, and then preliminary theory is developed to explain the hypothesis. The scientist then goes about designing an isolated experiment to analyze the theory. After collecting the results from the experiment, the scientist appends, refines, or rejects the preliminary theory. The experimental data collection then leads back to again observing nature and the cycle restarts [73]. While observations and theory can take considerable effort, this process is commonly stalled in the experimentation phase because of lack of resources like molecular biology tools to instantly measure particular interactions in living cells with a given accuracy. Since the dawn of the computer revolution, scientists now have another tool at their disposal in addition to constantly evolving experimental techniques. This tool is numerical simulation. Numerical simulation has proven quite effective, even dominant in certain types of research, such as astrophysics. Numerical simulation, although in its infancy in the field, has proven to give valuable insight into various functions vital to life including lipid metabolism and cell signal transduction [3, 98]. The use of computers and numerical science techniques in the life sciences has been predicted for quite sometime but it has only recently begun to take hold as a method of accelerating discovery

[52, 45].

Even with recent substantial advances in data management, genomics, and robotics, the discovery of new pharmaceutical agents has not accelerated over the last few years. Although the drug industry may be approaching a saturation point for single targeted drugs, simple drug treatments that effectively target a control network still hold a great deal of promise [7]. Even though biological network analysis is seldom used as inspiration for drug targets, its limited use has already enhanced the treatment of certain diseased states, most notably HIV. Mathematical simulations of these complex networks provide pharmaceutical researchers a means of linking the biochemistry of a reaction pathway to not only the resulting healthy phenotype but also to the dysfunctional disease state [13, 77, 67]. While still in its early years, the principles of systems biology could have profound effects leading to more hypothesis-driven research in drug discovery [45].

1.1.2 Previous Studies in Systems Biology

For years, biological control systems have been analyzed with control theory and recent studies [96, 97] are continuing this trend. Biological systems have shown both feed-forward and feedback control mechanisms. Several studies have demonstrated that bacteria chemotaxis depends on an integral feedback [2, 100]. With just minimal knowledge of kinetic parameters, biological control models allow not only a mechanism to show the response of biological network under different conditions but also a method of testing potential modifications to produce desirable effects [18]. The true difficulty in this approach comes in the location of equation specific kinetic parameters but can successfully be overcome with the use of analytical derivations and parameter estimation techniques [88]. In order to accurately simulate the sphingolipid metabolism pathway, a vast number of kinetic constants and reaction order constants must be acquired. Although equation-specific kinetic parameters are rarely available, a rough estimate can be obtained numerically since most systems are inherently robust and slight deviations from the optimum constants produces a minimum effect [2, 100]. Studies have shown biological control systems rely much more on the

structure of their network than the value of specific kinetic parameters. Even though a preliminary network model can usually be derived from existing data, a series of well-designed experiments with sufficient coverage can greatly enhance the model [46].

Biological metabolism and regulatory networks are amazingly robust against certain perturbations, but this robustness often does come at a price. The well-engineered systems, including biological ones, can be made highly resistant of expected variations but are often left very susceptible to slight unknown perturbations [12, 11]. Biological networks such as cancer cells have high resistance to known perturbations and are also likely exhibit extreme sensitivity to certain parameters [46, 47]. After performing system identification, a process of mathematically describing a network in terms of mathematical relationships and parameters, simulation of the system under a wide variety of hypothetical conditions can be preformed and analyzed to find the situations that induce a significant change in the systems behavior. Assessing the simulation's instability regions, conditions that produce large changes, and finding the sensitive parameters could potentially yield insight into non-intuitive treatment options for robust cancer cells [47]. The one caveat is that system identification can be exceedingly difficult in complex systems like the intracellular signaling regulation of cancer cells [31].

Metabolic Engineering has recently embraced the new opportunities afforded by recombinant DNA techniques to redirect metabolite towards desired products by accelerating or bypassing pathway bottlenecks [6, 75]. Initially, the shift towards desired projects was thought to be possible using two primary methods, knocking out enzyme function or inserting novel enzyme coding regions into the host's genome [83]. These pursuits failed to give the expected results. The principle flaw with these approaches concerns the overlooking of the complexity and rigidity of biological networks, which is vital to the organisms survival. This type of response is inline with what in seen in other fields such as microbes becoming resistant to antibiotics after prolonged exposure. Several early works cautioned the use of these methods as a perfect solution and recent studies in systems biology have confirmed it [2, 89, 100].

Several distinct yet mathematically related approaches have been applied in silco to

various metabolism pathways. The earliest method implemented basic chemical reaction engineering principles to link the flow from given substrates to potential products [55, 56, 57, 54, 58, 59]. The major advantage of this method is its hierarchical modular structure, but it may be too simplistic because it assumes that reactions can be broken down into single steps and then systematically reconstructed to get the final pathway. The use of a system of linearly independent basis vectors in flux space is another attempt to mathematically characterize metabolism reaction sequences [80]. This method is scaleable to increase the simulations scope but may fail to accurately account for nonlinear behavior, potential feedback mechanisms, and enzymatic reaction mechanisms. A third mathematical representation is one made from a series of elementary flux modules or basic reaction modules [83]. A network built on this method can be decomposed into subsystems that exert limited control on each other [82]. This modeling framework until recently could only make steady state assumptions, which is a major disadvantage since dynamic behavior resulting from sudden changes in conditions could explain how known signaling pathways are triggered in response to stimuli. The fourth type of mathematical expression commonly applied to metabolism is a power law representation known as the S-system [92, 94]. The S-system contains modules each consisting of power law differential equations that accurately account for the nonlinear behavior of biological regulatory network [93]. This modeling framework provides a complex differential equation-based model by linearizing the nonlinear system in logarithmic space. Although the S-System is not widely accepted among enzymologists and biochemists, the model's simple structure that allows feedback mechanisms to easily be included and insight to quickly be obtained is advantageous before employing more complex traditional models, such as Michaelis-Menten.

1.2 Sphingolipid Basics

In addition to simply picking an underlying chemical reaction formulation, systems biology research also necessitates picking a suitable test case. Here we have chosen a test case associated with a normal and abnormal state of sphingolipid metabolism because of availability

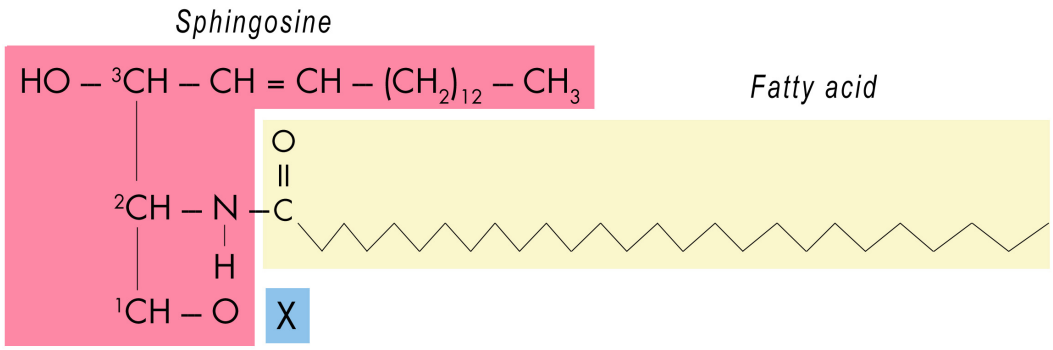

Figure 1: Sphingolipid Structure: All sphingolipids contain the following three basic components: one molecule of the long-chain amino alcohol sphingosine (shown in pink), one molecule of long chain fatty acid (shown in yellow) and some version of a polar head group (shown in the blue section).

of suitable high throughput data collection [62] and cell models. Before discussing the exact numerical equations that make up sphingolipid metabolism networks a small section of background information on sphingolipid functions, chemical structures, and synthesis process is appropriate. Figure 1 shows the three basic entities that when combined constitutes a sphingolipid species [50].

All complex sphingolipids contain the following three basic components: one molecule of the long-chain amino alcohol sphingoid base, one molecule of long chain fatty acid and some version of a polar head group. Although sphingolipid species can vary in any of the three pieces previously mentioned, serine most often combines with the fatty acid Palmitoyl-CoA in human cells to form the precursor that eventually becomes the sphingosine species in the Figure 2. Stearoyl-CoA is also a possible sphingosine precursor, but this class of sphingolipids is typically only found in brain tissue. The causes and effects of altering the composition of the sphingosine backbone is largely unknown but is thought to play some role in the aging process. Sphingolipids made of fatty acids of different lengths and different degrees of saturation can be produced through cellular signaling and environmental pressures. The long term effects on cells that produce a different composition of sphingolipids is largely unknown and could be a key indicator of progression of various diseases. The polar head group can be a variety of substances, such as sugars or phosphate groups, linked using glycosidic linkage or phosphodiester bond that attaches the sphingoid base core to

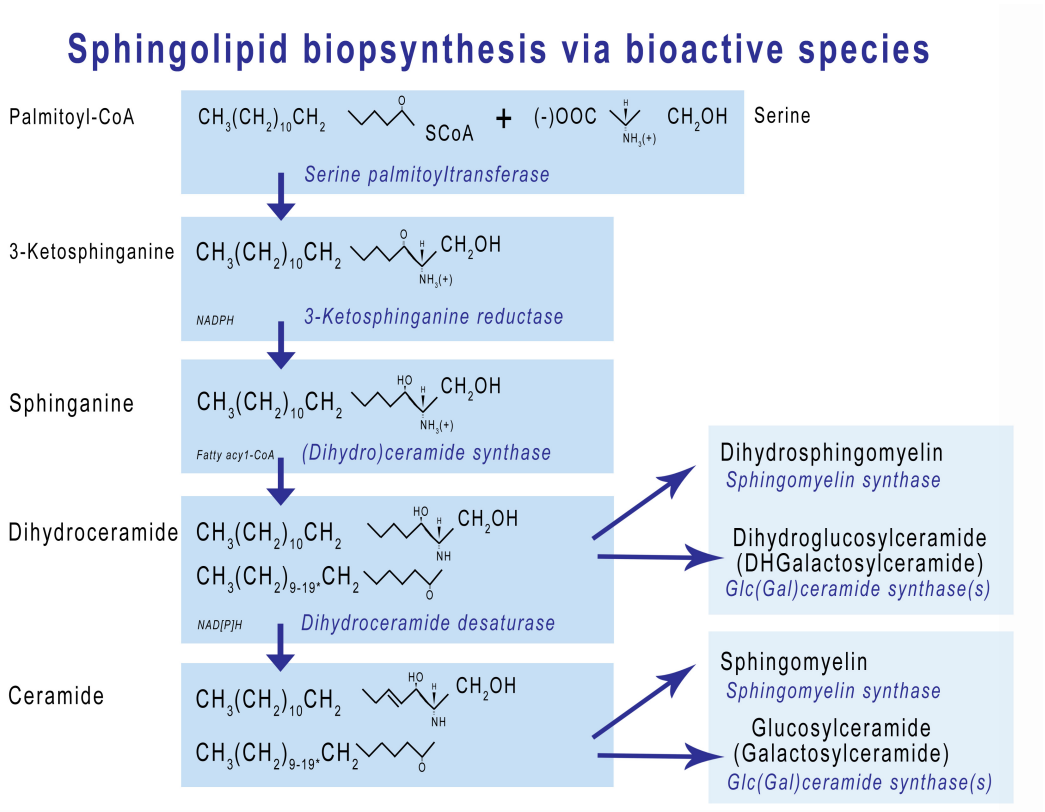

Figure 2: Sphingolipid Pathway Map: The sphingolipid de novo synthesis pathway is presented in the above figure [61].

the polar head group [1].

One of the biggest advantages of choosing sphingolipid metabolism as the first test case for a dynamic pathway algorithm is that the variety of species discussed in the previous paragraph are all chemically constructed from a common set of reactions in which a unique series of these reactions gives rise to a unique product. This feature can be best illustrated by presenting the sphingolipid metabolism pathway, which is shown in Figure 2 [61]. Another advantage is the potential therapeutic targets contained in the pathway. One potential therapeutic target in the sphingolipid metabolism system is to cause the accumulation of the species ceramide, which is the precursor to all other types of sphingolipids. Excessive ceramide accumulation has been shown to play a role in apoptosis [87]. An accurate model that describes the sphingolipid metabolic network is useful in determining the role of sphingolipids in complex diseases and also a first step towards accelerating the potential use of ceramide and other sphingolipid species as molecular targets fighting these complex

diseases such as cancer.

Previous studies that presented models of sphingolipid metabolism [3, 4] focused on gathering literature information to collect *in vitro* determined parameters for kinetic models of these reactions. This literature information was then used to derive a model for each of the considered reactions. The literature was also scanned for potentially important regulatory interactions. While literature is certainly the gold standard of available information, critical information required to build models is often difficult to obtain or be based on *in vitro* classifications in very different organisms. This study considers the relationships between six metabolites four of which that were not included in the published yeast model [4].

1.3 Mathematical Models of Enzymatic Reactions

Mathematical expressions that describe enzyme kinetics are certainly not new. Michaelis and Menten proposed what is still the most dominate expression in 1912. The late 1960s and 1970s saw the introduction of more complicated binding models as seen in [70]. Not much in concepts of enzyme kinetics has changed since the 1970s but [51] presents an updated comprehensive review of the available models both old and new. The mathematical expressions used in various parts of this work are presented in this section. The following models are presented: zero order kinetics, mass action kinetics, reversible mass action kinetics, Michaelis-Menten kinetics, reversible Michaelis-Menten kinetics, the kinetic UniUni mechanism, generalized mass action kinetics, reversible generalized mass action kinetics, and the S-system. If a more detailed derivation of any of the formalisms is required beyond the brief descriptions presented here, the noted references should be consulted.

1.3.1 Zero Order Kinetics

Zero order kinetics is the name given to a mathematical description of a chemical reaction where the flux from the reactant to the product is independent of both the reactant and product concentration. It is called zero order because all of the reaction orders of reactants and products are zero. This simplifies to give a constant flux of substrate being converted to product. Equations 1 – 5 shows a typical reaction of A converting into P modeled by a

Zero Order Reaction.

$$A \xrightarrow{k_f} P \tag{1}$$

$$Flux = k_f[A]^0[P]^0 \tag{2}$$

$$Flux = k_f = Constant \tag{3}$$

$$\frac{dP}{dt} = Flux \tag{4}$$

$$\frac{dA}{dt} = -Flux \tag{5}$$

1.3.2 Mass Action Kinetics

Mass action kinetics has the same form as zero order kinetics but the reaction orders under mass action have values that reflect their stoichmetric constants instead of being zero. This kinetic model has been used for the investigation of many biological reactions including ligand-receptor binding [49, 33]. A mass action model describing a molecule of A combining with two molecules of B to produce P is presented in Equations 6 – 10.

$$A + 2B \xrightarrow{k_f} P \tag{6}$$

$$Flux = k_f[A]^1[B]^2 \tag{7}$$

$$\frac{dP}{dt} = Flux \tag{8}$$

$$\frac{dA}{dt} = -Flux \tag{9}$$

$$\frac{dB}{dt} = -2Flux \tag{10}$$

1.3.3 Reversible Mass Action Kinetics

Chemical reactions sometimes are bidirectional meaning that under certain conditions it possible for the reactants to become the products as well as the products reverting back to the reactants. These types of reactions often reach equilibrium, where the forward production of products is equal to the reverse production of reactants. Reversible mass action kinetics is essentially the same as modeling a reaction with a mass action model in both the forward and reverse direction. A reversible mass action model describing a molecule of A combining with two molecules of B to produce a molecule of P and also

a molecule of P converting into a molecule of A and two molecules of B is presented in Equations 11 – 16.

$$A + 2B \overset{k_f/k_r}{\rightleftharpoons} P \tag{11}$$

$$ForwardFlux = k_f[A]^1[B]^2 \tag{12}$$

$$ReverseFlux = k_f[P]^1 \tag{13}$$

$$\frac{dP}{dt} = ForwardFlux - ReverseFlux \tag{14}$$

$$\frac{dA}{dt} = ReverseFlux - ForwardFlux \tag{15}$$

$$\frac{dB}{dt} = 2ReverseFlux - 2ForwardFlux \tag{16}$$

1.3.4 Michaelis-Menten Kinetics

Michaelis-Menten kinetics is the most popular model of enzyme reaction models in biochemical literature. This reaction model uses several mass action processes along with the steady state assumption on the enzyme levels to produce a model capable of displaying characteristics like enzyme saturation. The first reaction is the substrate and enzyme reacting via a reversible mass action process to produce an enzyme-substrate complex. This enzyme substrate complex then can undergo an irreversible mass action process to become product and free enzyme. The conversion of a substrate, S, to product, P, in the presence of enzyme, E, is presented in equations 17 – 22. K_M is the Michaelis constant, V_max is the maximal reaction velocity, and E_0 is the total amount of enzyme present.

$$A + E \overset{k_1/k_{-1}}{\rightleftharpoons} ES \overset{k_2}{\rightarrow} E + P \tag{17}$$

$$Flux = \frac{V_{max}S}{S + K_M} \tag{18}$$

$$V_{max} = k_2[E_0] \tag{19}$$

$$K_M = \frac{k_{-1} + k_2}{k_1} \tag{20}$$

$$\frac{dP}{dt} = Flux \tag{21}$$

$$\frac{dS}{dt} = -Flux \tag{22}$$

1.3.5 Reversible Michaelis-Menten Kinetics

Reversible Michaelis-Menten kinetics is the same as the original Michaelis-Menten formalism except the enzyme-substrate complex to product plus enzyme reaction is considered reversible. This minor switch leads to a rate law that is very similar to the original formalism and is derived for the same substrate to product enzymatic reaction as before [19]. This rate law is shown in equations 23 – 30.

$$A + E \overset{k_1/k_{-1}}{\rightleftharpoons} ES \overset{k_2/k_{-2}}{\rightleftharpoons} E + P \tag{23}$$

$$Flux = \frac{K_S E_0 S - K_P E_0 P}{1 + \frac{S}{K_{MS}} + \frac{P}{K_{MP}}} \tag{24}$$

$$K_{MS} = \frac{k_{-1} + k_2}{k_1} \tag{25}$$

$$K_{MP} = \frac{k_{-1} + k_2}{k_{-2}} \tag{26}$$

$$K_S = \frac{k_1 k_2}{k_{-1} + k_2} \tag{27}$$

$$K_P = \frac{k_{-1} k_{-2}}{k_{-1} + k_2} \tag{28}$$

$$\frac{dP}{dt} = Flux \tag{29}$$

$$\frac{dS}{dt} = -Flux \tag{30}$$

1.3.6 The Kinetic UniUni Mechanism

While combustion reactions are primarily driven by the random collison of molecules, an adsorption process primarily drives enzymatic reactions [51]. An enzymatic reaction can be defined by the process of the single substrate binding to free enzyme, which reversibly forms the enzyme-substrate complex that is then reversibly converted to the product-enzyme complex, which can be reversibly decomposed into the product and free enzyme. This type of reaction is often written as equation 31 [44].

$$E + S \overset{k_1/k_2}{\rightleftharpoons} ES \overset{k_3/k_4}{\rightleftharpoons} EP \overset{k_5/k_6}{\rightleftharpoons} E + P \tag{31}$$

The rate equation of product formation would be written as equation 32 [44].

$$\frac{dP}{dt} = k_5[EP] - k_6[E][P] \tag{32}$$

When the above equation is multiplied by the identity $\frac{[E_0]}{[E]+[ES]+[EP]}$ the resulting equation is given by equation 33.

$$\frac{dP}{dt} = \frac{k_5[EP][E_0] - k_6[E][P][E_0]}{[E] + [ES] + [EP]} \tag{33}$$

The enzyme species in the above equation can be substituted by solving for them in terms of rate constants, product, and substrate equations. The first step is to rewrite the chemical equation in terms of these enzyme containing species, which is presented in equation 34.

$$E \overset{k_1 S/k_2}{\rightleftharpoons} ES \overset{k_3/k_4}{\rightleftharpoons} EP \overset{k_5/k_6 P}{\rightleftharpoons} E \tag{34}$$

If equation 34 is broken into terms representing enzyme species, the resulting combinations describe the particular enzyme species in terms of the kinetic rate constants, product concentration, and substrate concentration. These representations can now be inserted into the product rate equation. The resulting equation is given in equation 35.

$$Flux = \frac{k_1k_3k_5[S][E_0] - k_2k_4k_6[P][E_0]}{k_2k_5 + k_2k_4 + k_3k_5 + (k_1k_3 + k_1k_4 + k_1k_5)[S] + (k_2k_6 + k_3k_6 + k_4k_6)[P]} \tag{35}$$

The rate of product production (and substrate depletion) is now defined in terms of the six kinetic parameters, substrate concentration, product concentration, and total initial enzyme concentration. Equations 35 and 36 show the equation for product accumulation and substrate depletion, respectively.

$$\frac{dP}{dt} = Flux \tag{36}$$

$$\frac{dS}{dt} = -Flux \tag{37}$$

1.3.7 Generalized Mass Action (GMA) Kinetics

Generalized mass action kinetics is an alteration of mass action kinetics that allows the reaction orders of each of the substrates and products to take on a value that does not necessarily directly reflect the stoichiometry of the reaction. This kinetic model has been used for the investigation of many biological reactions especially in the metabolism studies. A comprehensive reference on the GMA and mathematical operations that can be utilized

with this form is given in [94]. A generalized mass action model describing a molecule of A combining with two molecules of B to produce P is presented in equations 38 – 42.

$$A + 2B \overset{k_f}{\rightarrow} P \tag{38}$$

$$Flux = k_f[A]^{g_1}[B]^{g_2} \tag{39}$$

$$\frac{dP}{dt} = Flux \tag{40}$$

$$\frac{dA}{dt} = -Flux \tag{41}$$

$$\frac{dB}{dt} = -2Flux \tag{42}$$

1.3.8 Reversible Generalized Mass Action Kinetics

Chemical reactions sometimes are bidirectional meaning that under certain conditions it is possible for the reactants to become the products as well as the products reverting back to the reactants. These types of reactions often reach equilibrium, where the forward production of products is equal to the reverse production of reactants. Reversible generalized mass action kinetics is essentially the same as modeling a reaction with a generalized mass action model in both the forward and reverse direction. A reversible mass action model describing a molecule of A combining with two molecules of B to produce a molecule of P and also a molecule of P converting into a molecule of A and two molecules of B is presented in equation 43 – 48.

$$A + 2B \overset{k_f/k_r}{\rightleftharpoons} P \tag{43}$$

$$ForwardFlux = k_f[A]^{g_1}[B]^{g_2} \tag{44}$$

$$ReverseFlux = k_f[P]^{h_1} \tag{45}$$

$$\frac{dP}{dt} = ForwardFlux - ReverseFlux \tag{46}$$

$$\frac{dA}{dt} = ReverseFlux - ForwardFlux \tag{47}$$

$$\frac{dB}{dt} = 2ReverseFlux - 2ForwardFlux \tag{48}$$

1.3.9 Reversible Generalized Mass Action Kinetics

In the S-System format each of the elementary fluxes are grouped into aggregate fluxes that represent the total influx and total outflux for each metabolite pool. The standard differential equation for the S-System can be written as Equation 49.

$$\frac{dX_i}{dt} = V_{+i} - V{-}i \tag{49}$$

Any S-System differential equation can be written in the above form. V_{+i} is the aggregated influx. V_{-i} is the aggregated outflux. If each of the aggregate fluxes is represented with a power law approximation, the general form shown in Equation 50 is derived.

$$\frac{dX_i}{dt} = \alpha_i \prod_{j=1}^{n+m} X_j^{g_{ij}} - \beta_i \prod_{j=1}^{n+m} X_j^{h_{ij}} \tag{50}$$

1.4 Optimization Method Background

In addition to the underlying mechanisms that are used to quantify enzymatic reactions, optimization methods are also vital to this work to take an initial guess for the parameters and iteratively improving the parameters to better fit the experimental data. The methods that have been solicited for this fitting task are Monte Carlo selection, nonlinear least squares using the Gauss-Newton Method, nonlinear least squares using the Marquardt-Levenberg method, a Nelder-Mead based method that uses randomly determined initial conditions, Simulated Annealing, and Genetic Algorithm. Each of the optimization procedures is explained in the following sections.

1.4.1 Monte Carlo Methods

Monte Carlo methods have been applied in many problems, most notably integration. Monte Carlo methods offer a random sample of the entire search space. The principle behind this tool is that a sufficiently large random sample will be to converge to accurately represent the entire search space. In addition to gathering population statistics in a much timelier manner than exhaustive brute force searching, sufficient Monte Carlo sampling allows a reasonable approximation for the global extremes but does not guarantee that the true global minimum will be found.

1.4.2 Nonlinear Least Squares using the Gauss-Newton Method

The Gauss-Newton method is an iterative method used to overcome difficulties in satisfying least squares that are presented when fitting an expectation function (i.e. the mathematical model which relates the parameters and independent variables) as closely as possible to the experimental data. Least Squares can be simply constructed with the following two statements.

1. First, find the point $\overline{\eta}$ on the expectation surface (the surface created by the mathematical model being fit to) that is closest to y (the vector containing actually experimental measurements).

2. Secondly, determine the parameter vector that corresponds to this closest point $\overline{\eta}$.

For a linear model, both of the above steps are straightforward. Step 1 is straightforward because the expectation surface is a plane of infinite extent, which allows the use of an explicit expression for the point on the plane closest to y. This expression is shown below in equation 51 [8].

$$\overline{\eta} = QQ^T y \tag{51}$$

Q in the above equation represents the orthogonal matrix produced from QR matrix decomposition. $Q^T$is the Q matrix transposed and y is the experimental data column vector. The second statement describing least squares with regards to a linear system can also be explicitly defined by a simple equation, which is given in terms of QR decomposition in equation 52 [8].

$$\overline{\theta} = R^{-1}Q^T\overline{\eta} \tag{52}$$

$\overline{\theta}$ is the parameter vector that corresponds to the closet point on the expectation surface,$\overline{\eta}$. R^{-1} in the above equation corresponds to the inverse of the upper triangular matrix created by QR matrix decomposition. Q^T again represents the transpose of the orthogonal matrix.

Accomplishing the two steps presented above describing the least squares process is exceedingly more difficult for the nonlinear version than the linear version with its explicit

formulas. The first step is difficult to achieve using a nonlinear system because the expectation surface is curved and at least has edges or is often of finite extent. The second least squares require is also difficult to uphold because the system maps easily in only one direction, from the parameter plane to the expectation surface (opposite of what item 2 requires). The Gauss-Newton method uses a linear approximation to the expectation function to iteratively improve the initial guess for the parameter plane until the difference between successive iteration is negligible. The process starts with a Taylor series expansion about the initial parameter plane. This expansion is represented below in equation 53 [8].

$$f(x_n, \theta) \approx f(x_n, \theta^0) + v_{n1}(\theta_1 - \theta_1^0) + v_{n2}(\theta_2 - \theta_2^0), \ldots, + v_{nP}(\theta_P - \theta_P^0) \tag{53}$$

θ in the above equation represents the true best fit parameters, θ^0 represents the initial guess at the best parameters, x_n is a vector containing the independent variables (in the sphingolipid metabolism system this is the concentration of each node species), and f is the expectation function, the function that relates the fitting parameters to the independent variables. The partial derivative of an independent variable with respect to a particular fitting parameter, generically θ_P, is given as the quantity v_{np}. The equation for v_{np} is given in equation 54 [8].

$$v_{np} = \frac{\delta f(x_n, \theta)}{\delta \theta_P}|_{\theta^0} \tag{54}$$

The V^0 matrix (initial derivative matrix) of size NxP (independent variables by the number of fitting parameters) is formed with elements v_{np}. Incorporating all independent variables the following general expression can be written as equation 55 [8].

$$\eta(\theta) = \eta(\theta^0) + V^0(\theta - \theta^0) \tag{55}$$

The initial residuals z^0 are defined in equation 56. The actual residuals $z(\theta)$ are described in terms of z^0 and δ, the difference between the actual best parameters, θ, and the initial guess for the best parameters, θ^0, is also shown below [8].

$$z^0 = y - \eta(\theta^0) \tag{56}$$

$$\delta = \theta - \theta^0 \tag{57}$$

$$z(\theta) \approx y[\eta(\theta^0) + V^0\delta] = z^0 - V^0\delta \tag{58}$$

The approximation of the true residuals, noted above, can now be used in traditional least squares methodology to find δ^0, the initial Gauss increment. The least squares process that is used to determine the δ^0, which is the increment applied to improve the initial parameter guess, θ^0, is shown below in equations 59 – 62. Again QR decomposition is used.

$$V^0 = Q_1 R_1 \tag{59}$$

$$\overline{\eta}^1 = Q_1(Q_1^T z^0) \tag{60}$$

$$R_1\delta^0 = Q_1^T z^0 \tag{61}$$

$$\theta^1 = \theta^0 + \delta^0 \tag{62}$$

After the initial parameter guesses have been incremented, the process restarts by calculating a new derivative matrix and repeating the above least squares routine to find the next Gauss increment, δ^1, which is again applied to the parameter vector to improve it. This process continues until δ^n is negligible [8]. This is a local parameter search because it depends on the first derivative. When the first derivative becomes essentially zero, the Gauss increment is also approximately zero, which calls for the routine to stop repeating.

1.4.3 Nonlinear Least Squares using the Marquardt-Levenberg Method

The Marquardt-Levenberg method is an incremental enhancement of the Gauss-Newton method presented previously. Near singularity of V, the derivative matrix, causes δ, the Gauss increment to be very large, which may move the iterative method into undesirable regions of the parameter space (imaginary regions). The V matrix can become singular when the columns of the matrix are collinear. Levenberg proposed the initial solution to the derivative matrix singularity problem in 1944. Levenberg alters Gausss formulation of the increment δ using equation 63 [8].

$$\delta(k) = (V^T V + kI)^{-1} V^T (y - \eta) \tag{63}$$

I in the above equation is the identity matrix, k is a conditioning factor, V again is the derivative matrix, y is the vector of measured values, and η is the closet point on

the expectation surface to y. The above reformulation of δ results from the least squares solution to a system with an altered derivative matrix, V_L, shown below in equations 64 – 68. Substituting V_L into the Gauss-Netwon yields the Levenberg Method of performing nonlinear least squares. This methodology is shown below the definition of V_L [8].

$$V_L = \begin{bmatrix} V \\ \sqrt{k}I \end{bmatrix} \tag{64}$$

$$V_L^0 = Q_{L1}R_{L1} \tag{65}$$

$$\overline{\eta_L}^1 = Q_{L1}(Q_{L1}^T z^0) \tag{66}$$

$$R_{L1}\delta_L^0 = Q_{L1}^T z^0 \tag{67}$$

$$\theta_L^1 = \theta^0 + \delta_L^0 \tag{68}$$

The above algorithm is an altered version of the Gauss-Newton method presented previously. All symbols represent the same symbols as the Gauss-Newton method except where differences are noted with a subscript L, meaning these symbols are redefined when using the Levenberg method. Although the Levenberg method does in fact solve the singularity problem with the derivative matrix in the Gauss-Newton method, it has its own deficiency in the fact that the Levenberg increment is variant under different scaling transformations. In order to provide an invariant increment under all scaling transformations, Marquardt proposed a minor alteration of the Levenberg method in 1963. Marquardt's increment inflates the diagonal of the derivative matrix by a factor of $1 + k$ instead of k as suggested by Levenberg. The Marquardt increment is shown in the equation 69 [8].

$$\delta(k) = (V^T V + kD)^{-1} V^T (y - \eta) \tag{69}$$

D in the above equation is the diagonal matrix with entries equal to the diagonal entries of $V^T V$. The Marquardt increment is given by following the Gauss-Newton process starting with an altered derivative matrix, V_M. First the equation, 70, describes how V_M is formed from the derivative matrix. The remaining equations detail how the Marquardt increment is applied to the Gauss-Newton process [8].

$$V_M = \begin{bmatrix} V \\ \sqrt{k}D^{1/2} \end{bmatrix} \tag{70}$$

$$V_M^0 = Q_{M1} R_{M1} \tag{71}$$

$$\overline{\eta_M}^1 = Q_{M1}(Q_{M1}^T z^0) \tag{72}$$

$$R_{M1}\delta_M^0 = Q_{M1}^T z^0 \tag{73}$$

$$\theta_M^1 = \theta^0 + \delta_M^0 \tag{74}$$

The above equations represent the Marquardt-Levenberg method for performing non-linear least squares [8]. The method is equally as computationally effective but much more robust than Gauss-Newton method. The Marquardt-Levenberg method still relies on a linear approximation and a first derivative, which makes the method tend to be equally satisfied with local and global minima. Therefore, searching a rough terrain (an expectation surface with many local minima) to find a global is difficult unless initial conditions can be approximated close enough to the global minimum to prevent entrapment in a local minimum.

1.4.4 Nelder-Mead Simplex Method for Function Minimization

The Nelder-Mead downhill simplex method is an efficient derivative-free way of determining the local minimum of function of multiple independent variables. The algorithm works to minimize an N -dimensional (containing N independent variables) cost function by first setting up an N-dimensional geometric figure consisting of $N + 1$ points. This geometric figure is called a simplex. The first point on the simplex is the initial guess of the minimum. The other points that make up the simplex, P_i, are determined by the equation 75 [72].

$$P_i = P_0 + \lambda \epsilon_i \tag{75}$$

P_0 represents the initial guess of the lowest point, λ in the above equation is constant representing the estimated characteristic length of the specific problem, $epsilon_i$ is specific unit vector of the set ϵ, which contains N unit vector in different directions. Once the initial simplex is set up, the algorithm first evaluates P_0 and each P_i in the cost function to determine which one produces the largest cost. The Nelder-Mead requires an additional cost function based on the experimental system, which is minimized. The cost function can be

defined as the sum of squared residuals, like Gauss-Newton and Marquardt-Levenberg, but is not limited to being dependent on the squared residuals. After the point with the largest cost has been determined, the algorithm undertakes a combination of three operations, reflection, contraction, or expansion to iteratively remove the worst point on the simplex. The first attempted operation for each of the iterations is always reflection. The reflection equation to determine a new simplex point, P^*, is given in equation 76 [64].

$$P^* = (1+\alpha)\overline{P} - \alpha P_h \tag{76}$$

α is the reflection coefficient, a positive constant, P_h is the point on the simplex that gives the highest cost, and $\overline{P}$ is the centroid of the remaining points on the simplex. P^* is then evaluated in the cost function to generate y^*. If y^* is between y_h, the value of the cost function at P_h, and y_l, the minimum cost of the points in the simplex, P^* is accepted as a replacement for P_h and the algorithm begins another iteration by determining the new P_h and repeating the same process. If $y^* < y_l$, a new minimum has been located and the algorithm will attempt to expand further in the direction of the new found minimum, which is done using the expansion equation given in equation 77 [64].

$$P^{**} = \gamma P^* + (1+\gamma)\overline{P} \tag{77}$$

P^{**} is the new point on the simplex after expansion, P^* is still the reflected point, $\overline{P}$ is still the centroid of other points on the simplex, and γ is the expansion coefficient, which is greater than unity and represents the ratio $[P^{**}\overline{P}]$ to $[P^*\overline{P}]$. If y^{**}, the cost function evaluation of P^{**}, is less than y_l, P^{**} replaces P_h and the algorithm restarts. If $y^{**} > y_l$, then expansion failed and P^* will replace P_h and the algorithm will restart [64]. If on reflecting P_h to P^*, y^* is still greater than the cost of all other points on the simplex, the point with less cost (between P^* and P_h) is assigned to be the new P_h and the algorithm enters into a contraction step. The contraction equation is give below by equation 78.

$$P^{***} = \beta P_h + (1-\beta)\overline{P} \tag{78}$$

The contraction coefficient, β, is a constant between zero and one that represents the ratio of the distance of $[P^{***}\overline{P}]$ to $[P_h\overline{P}]$. P^{***} is accepted as a replacement for P_h and the

algorithm restarts unless y^{***} is greater than lesser of y^* and y_h, which therefore results in a failed contraction. A failed contraction causes all of the points on the simplex to be replaced according to the following equation, 79 [64].

$$P_i^{new} = \frac{P_i + P_l}{2} \tag{79}$$

The point on the simplex with the lowest cost, P_l, is added to each point on the simplex and then divided by 2 thus moving the entire simplex closer to the best point and the algorithm restarts with the new simplex. The Nelder-Mead downhill simplex algorithm continues to apply reflections, expansions, and contractions until the cost is minimized. The algorithm exits when the change in the cost does not exceed a preset tolerance [64]. The Nelder-Mead downhill simplex method is an effective minimization scheme because of its simplicity and effectiveness. The method has such advantages such being able to converge to a correct minimum even if the initial simplex is situated over a region with two local minima (two valleys) and being able to apply constraints to certain values. The algorithm also has several disadvantages such as being dependent on the initial conditions because the method not being able to converge a significant distance outside of the initial simplex.

1.4.5 Simulated Annealing

Simulated Annealing is an optimization procedure based on a natural optimization procedure in the area of thermodynamics. If a metal is heated to a temperature above its melting temperature, the individual metal atoms are in a state of highly excited random motion. When these excited atoms are gradually cooled, a process called annealing, the atoms begin to lose thermal mobility as the energy states continue to be lowered. As the cooling process is progressing, the system may enter various meta-stable states (local minima) only to gain energy and progress to the lowest possible energy state. When a sufficient cooling cycle is applied to a molten metal, nature is always able to settle into the lowest energy state, a global minimum, which is a highly structured crystal. The general optimization method and the approximate natural mathematically depend on the Boltzmann Distribution, which

is defined with the equation 80 below [81].

$$p(E) = \frac{1}{kT} \exp(\frac{-E}{kT}) \tag{80}$$

E in the above equation represents a particular energy state, k is the Boltzmann constant, T is the temperature of the system, and $p(E)$ is the probability of energy state E at temperature T. Suppose a n x 1 vector x contains the configuration of the individual atoms (the value of adjustable parameters in the general optimization case) and $f(x)$, the objective function, computes the energy level E (relative cost in the general case). The annealing process effectively performs unconstrained optimization [81]. If Δx is defined as a vector size x that represents a random change in the values of x. Supposing a k analogous to the Boltzmann constant, and an artificial temperature T, the distribution of Δx could be represented by the following Gaussian distribution with a zero mean [81].

$$p(\Delta x_j) = \frac{1}{\sqrt{2\pi}T} \exp(\frac{-\Delta x_j}{T^2}) \tag{81}$$

The above probability distribution has a standard deviation of T, which means for the initial large values of T the random changes will be large but will decrease in magnitude as the temperature is gradually reduced. If the random change in x, Δx, is applied to an initial guess, x^0, the resulting change in the objective function, Δf, is defined by the equation 82 [81].

$$\Delta f = f(x^0 + \Delta x) - f(x^0) \tag{82}$$

If $\Delta f < 0$, this represents an improvement in the value of the objective function so $x^0 + \Delta x$ is accepted as the new best point, x^1. However, even if $\Delta f \geq 0$, the increment can still be accepted if the probability of Δf, determined by equation 83 below, is greater than uniform random number generated on the interval zero to $\frac{1}{kT}$ [81].

$$p(\Delta f) = \frac{1}{kT} \exp(\frac{-\Delta f}{kT}) \tag{83}$$

The acceptance of a detrimental step, a step that increases the value of the objective function, allows the algorithm to escape local minima. If neither a true accepted step nor a detrimental accepted step are accepted, the process continues to generate a new Δx until a

step is accepted creating an incremented x value, x^1. As the temperature is being decreased on a predefined cooling schedule, the process of incrementing x continues until an iteration maximum is reached or x is not moving beyond a predefined tolerance [41].

Simulated Annealing is quite different from the nonlinear parameter estimation algorithms discussed thus far in that it does not aggressively and irreversibly attempt to find the minimum of the objective function, a cost function made from comparing experimental data to a specific point on the expectation surface. The reversible property of simulated annealing allows the method to escape local minima by enabling a random chance that the value of the objective function can actually increase. Since the algorithm has the ability to escape local minima, Simulated Annealing can be used with a wide variety of initial conditions on rough terrain (an expectation surface with many local minima) and with the proper cooling schedule and enough computational time will always approach the same minimum value of the objective function. This characteristic of the algorithm has been exploited to solve problems that were previously thought to be unsolvable. The famous traveling salesman problem is an example of the type of combinatorial optimization problem Simulated Annealing can solve of size N on the order of N to some small power instead of the exp(*constant* x N), which is required for a brute force solution [72]. The traveling salesman problem and the application of simulated annealing to solve it are described in [68].

Although Simulated Annealing has many advantageous qualities already mentioned, these qualities come with a high computation cost because of the passive nature of algorithm. The efficacy of the algorithm is also very dependent on the cooling schedule set at initialization. The cooling cycle has to be "sufficiently slow" to mimic that of nature but "sufficiently slow" can vary greatly between similar problems.

1.4.6 Genetic Algorithm

The genetic algorithm, initially conceived by John Holland, is population-based routine rooted in the evolution theory of Charles Darwin [36]. The genetic algorithm mimics the principles of natural selection through a series of operators defined in computer code.

Selection, crossover, mutation, and inversion were the basic operators employed in Hollands original formulation. Most current genetic algorithm implementations, including the ones discussed in this work, only employ the selection, crossover, and mutation operators [63].

The selection operator decides which members of the population are most likely to reproduce into the next generation. The more fit an individual, or in the case of optimization the lower the cost of the specified parameters, the more likely it is able to have its offspring be represented in the next generation (step in algorithm progression). The GA crossover operator is analogous to the principle of recombination in genetics. The simplest crossover operator divides the binary chromosomes into two sections at a randomly chosen place. One set of the complimentary sections from each parent is exchanged giving rise to two unique children chromosomes. More complicated crossover operators divide the binary chromosomes into a variable number of pieces by assigning a crossover probability for any section of the chromosome. The mutation operator, which has a low probability for occurrence, simply flips a particular bit in the binary chromosome. Typically, the crossover and mutation operators have been implemented in series but some more recent implementations actually implement these operations in parallel thus insuring a percentage of the new population is independently created from each operator.

A scheme for a simple genetic algorithm starts with generating an initial population of a certain size. If no prior knowledge or preliminary method can generate a reasonable initial population, the initial population is most often randomly generated. The fitness of each individual in the initial population is then calculated. Next, the selection method is employed, usually with replacement, to generate the number of parental pairs necessary to reproduce a population of the same size. The crossover operator then recombines the parental pairs to produce unique offspring. After the offspring are produced, the mutation operator randomly alters bits to slightly change members of the offspring population. The offspring population then replaces the parent population and the process restarts by first evaluating the fitness of each member of the new generation. This population is passed through the series of operators again to produce another new population. This process continues until some exit criterion is met, most often after a certain number of generations

has been exhausted [29, 63].

CHAPTER II

CONSTRUCTION OF A MODELING FRAMEWORK

This chapter will discuss the progress in building a simulation model that not only attempts to overcome the deficiencies of its predecessors but also produce a large-scale, biochemically relevant model that fits within the framework of established enzyme kinetics literature. The work is presented in phases in chronological order to emphasize the progression from elementary chemical reactions to parameter estimation in a simulated sphingolipid metabolism system.

2.1 General Algorithm Design Phase 1

The first phase of algorithm development involves creation of three distinct functions and establishing a workflow relationship to connect the individual pieces cohesively. The individual functions in this phase were an optimization routine, a cost function, and a system of ordinary differential equations that govern the accumulation or depletion of each particular sphingolipid species or precursor in the network being studied. Two sphingolipid metabolism systems, each having six time points with multiple replicates at each time point were used in this first phase. The particulars of these systems will be discussed in the following two sections that describe the lowest level of proposed algorithm, the system of ordinary differential equations. Figure 3 shows the relationship between the three primary functions in the phase one algorithm design.

2.1.1 Building Nonlinear Rate Functions

The initial six-node system was composed of a group of sphingolipids all containing the same fatty acid chain attached to the sphingosine backbone. In other words, the same sphinganine precursor is combined with serine to produce a sphinganine that is common to all species. The common sphinganine is then combined with a specific fatty acid (C16) to produce a dihydroceramide species that serves as the root of the six-node system. This

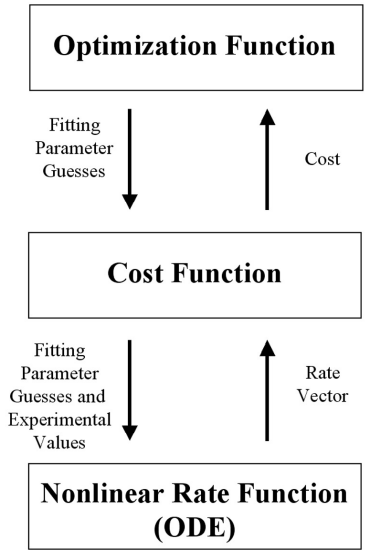

Figure 3: High Level Algorithm Design Phase 1: The relationship between the three primary components of the design are shown as well as the information that is passed between each of them.

dihydroceramide species can be converted in three other species bringing the network size to four. The reactions of the dihydroceramide species include the following: the enzymatic replacement of the hydrogen X group (see figure 1) with a phosphocholine to produce dihydrosphingomyelin, the enzymatic replacement of the hydrogen X group (see figure 1) with a glucose to produce dihydroglucosylceramide, and a desaturation reaction that inserts a double bond in the sphingosine backbone to produce ceramide (see figure 2). The first two reactions in this proposed pathway are considered terminal with a parameter that allows for "leak", which is production or comsumption by a chemical species that falls outside the indicated network. This concept of "leak" will be discussed in further detail when the governing differential equations are presented later in this section. The third of the above reactions produces ceramide, which is able to be further converted into different species by replacing the hydrogen X group (see figure 1) with either a phosphocholine to produce sphingomyelin or a glucose to produce glucosylceramide. The addition of the sphingomyelin and glucosylceramide nodes constitutes the entire 6-node system. The sphingomyelin and glucosylceramide nodes are also terminal nodes analogous to the dihydrosphingomyelin and

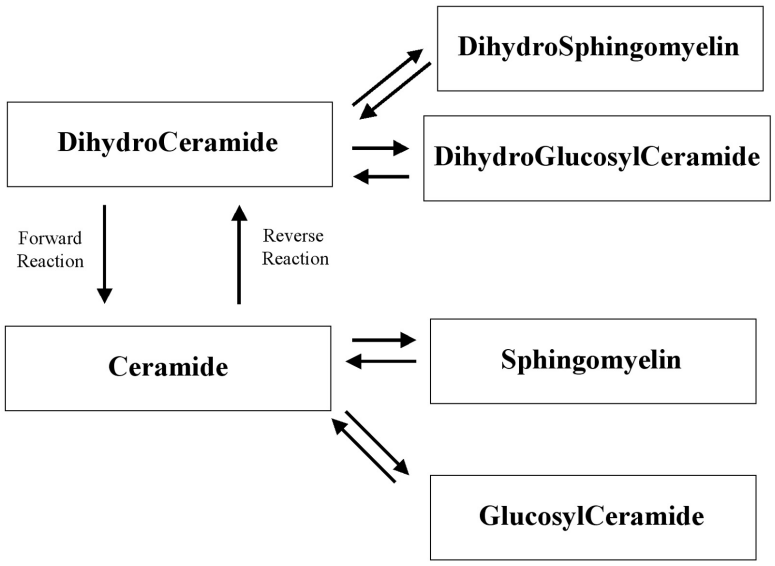

Figure 4: Six Node Sphingolipid Metabolism System considered in Phase 1: The chemical names in the model are given for each of the metabolites. The modelled metabolic network is composed of six dependent variables. Each of the five reversible fluxes are placed on the map as numbered arrows.

dihydroglucosylceramide nodes. The figure 4 is a graphical representation that shows the relationship between the nodes as discussed in the preceding paragraph. Note that all reactions are reversible.

Now that the preliminary network topology has been established, the general equations governing the interactions of the above nodes can be investigated. As noted before the dihydroceramide to ceramide reaction is a desaturation reaction, which is loosely defined as a catalytic process in fatty acids that replaces two carbon-hydrogen bonds by substituting a double carbon-carbon bond for a single one. This requires several reactants besides the dihydroceramide species including hydrogen ions, oxygen, and nicotinamide adenine dinucleotide phosphate (NADP) [50], which could not be measured to the same accuracy as the primary species at the time when equations governing this model were constructed. Since measurements for all of the species in the desaturation reaction could not be obtained, the forward reaction is assumed to be driven only by the concentration of dihydroceramide and the reverse reaction is likewise only driven by the concentration of ceramide. This assumption is valid in this case because the cellular concentration of the other reactants

is in large excess compared to the sphingolipid species thus making the sphingolipids the rate limiting reactants in the respective forward and reverse reactions. The other four interactions between nodes involve the addition (and the reverse) of a molecular species being added to the dihydroceramide or ceramide species in order to produce a different sphingolipid. The forward and reverse reactions for all of these interactions were again made dependent only on the concentration of the particular sphingolipid species instead of all reactants. This assumption is not as strong as in the case of the desaturation reaction because the concentration of glucose or phosphocholine may not be in massive excess in the sections of the cell involved in the metabolism process, which could result in a strong dependency on the concentration of either phosphocholine or glucose.

Although the assumptions outlined previously sufficiently provide the reversible reaction pieces necessary to use mass action kinetics discussed in Section 1.3, a slight correction must be applied to allow the system to exist in the context of a larger biochemical network (the cell) in which there is some ambiguity as to what other cellular reactions affect this pathway. This slight correction is the "leak" term discussed previously in this section. The "leak" term can either increase the concentration of a particular species or decrease it. A "leak" term that has a positive effect on the accumulation of a species could be caused by the breakdown of other sphingolipid species not in the 6-node pathway. A "leak" term that reduces the accumulation of a particular species could be representative of a reaction that consumes the sphingolipid species in order to build a more complex sphingolipid species. An example of this reduction of a particular species is the consumption of glucosylceramide in order to make lactosylceramide. A few important notes on the "leak" term in this phase of development are that only a single "leak" term is included (this term effectively models the difference between outside consumption and production) and the outside production or consumption is assumed to be constant in time and independent of all species concentrations. Although these assumptions, especially the independent of all species concentrations one, are not a particularly strong, they serve as a simple starting ground in which more complex schemes can be built. After all the assumptions governing the reversible reactions and the "leak" term have been incorporated into a mass action kinetics scheme, balance equations

84 – 89 are produced for the network map in figure 4.

$$\frac{dC_{DH}}{dt} = M_{DH} - k_1 C_{DH}^{\alpha_1} + k_2 C_C^{\alpha_2} - k_3 C_{DH}^{\alpha_3} + k_4 C_{SMDH}^{\alpha_4} - k_5 C_{DH}^{\alpha_5} + k_6 C_{GCDH}^{\alpha_6} \tag{84}$$

$$\frac{dC_{SMDH}}{dt} = M_{SMDH} + k_3 C_{DH}^{\alpha_3} - k_4 C_{SMDH}^{\alpha_4} \tag{85}$$

$$\frac{dC_{GCDH}}{dt} = M_{GCDH} + k_5 C_{DH}^{\alpha_5} - k_6 C_{GCDH}^{\alpha_6} \tag{86}$$

$$\frac{dC_C}{dt} = M_C + k_1 C_{DH}^{\alpha_1} - k_2 C_C^{\alpha_2} - k_7 C_C^{\alpha_7} + k_8 C_{SM}^{\alpha_8} - k_9 C_C^{\alpha_9} + k_{10} C_{GC}^{\alpha_{10}} \tag{87}$$

$$\frac{dC_{SM}}{dt} = M_{SM} + k_7 C_C^{\alpha_7} - k_8 C_{SM}^{\alpha_8} \tag{88}$$

$$\frac{dC_{GC}}{dt} = M_{GC} + k_9 C_C^{\alpha_9} - k_{10} C_{GC}^{\alpha_{10}} \tag{89}$$

2.1.2 Cost Function

The cost function and optimization method are intertwined in the cases where nonlinear regression was used to fit the data. In the intertwined case the cost function is the least residual squares. In cases where the optimization method and cost function are separate, the fitting method is the sum of absolute residuals. The method of calculating cost in instances where the optimization method and cost function are separate is given by equations 90 – 93.

$$Cost1 = |\dot{C}_{DH}^{DATA} - \dot{C}_{DH}^{CALC}| + |\dot{C}_{SMDH}^{DATA} - \dot{C}_{SMDH}^{CALC}| \tag{90}$$

$$Cost2 = |\dot{C}_{GCDH}^{DATA} - \dot{C}_{GCDH}^{CALC}| + |\dot{C}_{C}^{DATA} - \dot{C}_{C}^{CALC}| \tag{91}$$

$$Cost3 = |\dot{C}_{SM}^{DATA} - \dot{C}_{SM}^{CALC}| + |\dot{C}_{GC}^{DATA} - \dot{C}_{GC}^{CALC}| \tag{92}$$

$$Cost = Cost1 + Cost2 + Cost3 \tag{93}$$

The above equations shows the experimental derivative of each species being compared to the calculated derivative of each species. The absolute values of these comparisons are summed to give the cost, which is reported to the optimization function. An important note is in the comparison of concentration derivatives a good deal of error is introduced by calculating the experimental derivative since the time scale is in hours. Although the concentration is thought to change on a much faster time schedule than hours, this limitation was accepted in this preliminary development stage.

2.1.3 Results and Discussion

The following section presents the results of the phase 1 system and discusses these results. Six cost function and optimization combinations were used to generate the following results. Gauss Newton and Marquardt-Levenberg algorithms were used with the standard least squares cost function. Simulated Annealing and Nelder-Mead, both discussed previously in section 1.4, were paired with the sum of absolute residuals cost function as presented in the cost function section, 2.1.2. Two simple additional methods were included as a baseline benchmark by which the more sophisticate methods can be judged against. These simple methods are linear regression and random selection.

2.1.3.1 Linear Regression

Least squares linear regression was the first parameter estimation method attempted due to its speed and simplicity. Linear regression is easily computed from the systems equation presented in the previous nonlinear rate function with all αs set equal to 1. After forming the respective X and Y matrices using the 30 experimental data points, a single line of Matlab code, $Param = inv(X^T X) * X^T * Y$, which very similar to the linear algebra process that can be done by hand, is required to solve for the best least squares fit. Note that when solving the linear system of equations X_1 and X_4 must be divided into X_{11}, X_{12}, X_{13}, X_{41}, X_{42}, and X_{43}, respectively. The six resulting graphs from the linear fit are presented below in Figure 5.

As Figure 5 clearly shows, linear regression on the entire system equation does not appear to give a quality fit. The system is dependent on other variables, or it is not linear. The absolute sum of residuals is approximately 4000 for this fit. The processing time required was less than 5 seconds. There is no variability in the output answer.

2.1.3.2 Gauss-Newton Method

Gauss suggested a method for fitting nonlinear equations that uses a linear approximation of expectation function to iteratively improve the initial guess towards the actual solution of the parameters. This method effectively does a Taylor Series expansion on the data matrix

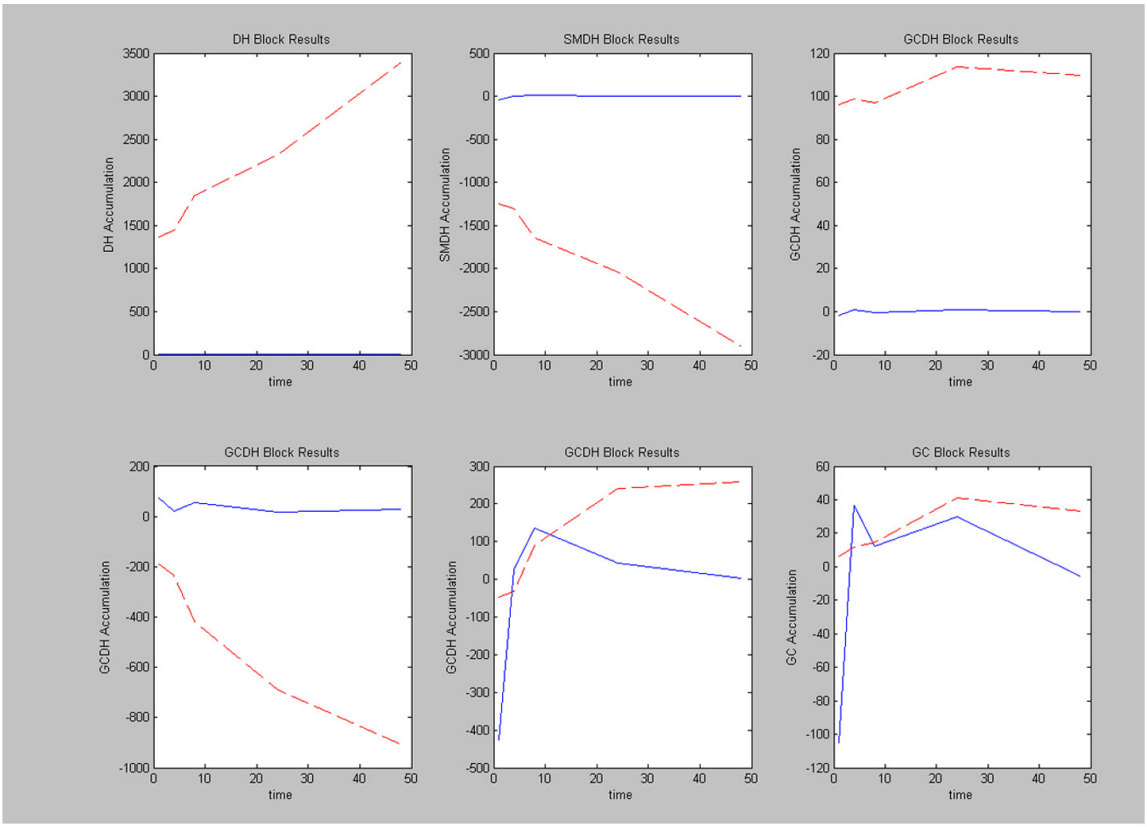

Figure 5: Linear Regression Fitting Results Phase 1: The blue solid lines represent the time derivative taken from actual data measurements. The discontinuous red line shows the line predicted by the linear regression parameters.

and ignores the higher order terms to get a linear approximation.

The Gauss-Newton method is available in both the Matlab statistics toolbox and the optimization toolbox. When the algorithm was employed to fit the six-node network system equation, parameters were found in roughly 10 seconds. Although the algorithm returned numbers, the fitted parameters were nonsense and caused the system equation to be NaN (not a number) at every evaluation point. The algorithm may be sensitive to initial conditions but that was not seen in this system because it never produced a reliable answer regardless of the start position.

Singularity of the matrix used in determining step size causes the step size to be rather sizeable when the matrix is nearly singular giving the Gauss-Newton method erratic behavior. The singularity of the derivative matrix arises because of the collinearity of its columns [8]. It should be noted that in the later stages of this project it was realized that singularity of this matrix might have been avoided if a mechanism was in place to prevent the species concentration from becoming negative. The derivative can in fact be negative (loss of accumulation) but negative concentration has no physical meaning.

2.1.3.3 Marquardt-Levenberg

The Marquardt-Levenberg (ML) is an alteration of the Gauss-Newton algorithm in which the step size is determined from a different mathematical variation of the derivative matrix. The step size of this algorithm is $delta(k) = (V^T V + kD)^{-1} V^T (y - \eta)$ where $y - \eta$ is the residual value using the current parameters, V is the derivative matrix, k is the conditioning factor, and D is a diagonal matrix whose entries are equal to the diagonal entries of $V^T V$ [8]. This algorithm is much more robust and seemed rather insensitive to initial conditions. The absolute sum of residuals for this fit was consistently around 1100 regardless of the number of iterations permitted. The computational time required for convergence of a random initial condition ML fitting is approximately 3 seconds. The results of this algorithm are presented in figure 6.

Figure 6 is closer to the actual data than the results produced from linear regression but it still does not capture the dynamic nature of the system. The reaction model could

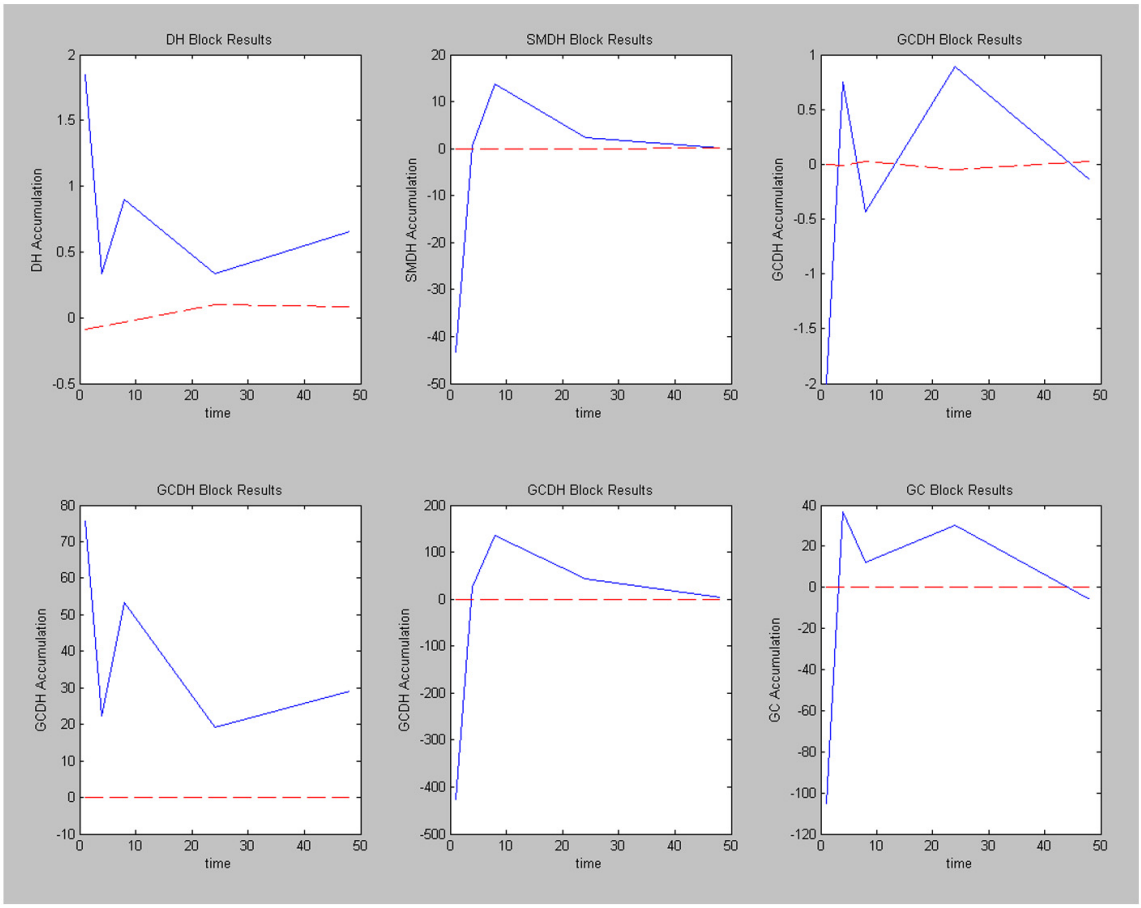

Figure 6: Marquardt-Levenberg Fitting Results Phase 1: The solid blue line represents the derivative calculated from actual experimental data and the red dashed line represents the values predicted by the parameters from an ML nonlinear fitting.

be inadequate to describe the system or the time points in which the experimental data were collected could cause instability in the derivative used to calculate the function used in fitting.

2.1.3.4 Randomized Nelder-Mead

The Nelder-Mead(NM) simplex (direct) search method has been identified as a very capable algorithm in finding local minima in multidimensional space [41]. The algorithm sets up an initial n-dimensional simplex (in three dimensions this is a triangle) and implements a set of conditions that shrink the simplex towards the local minimum [76]. This method characterized by an average computational speed (3 hours for 5,000 initial conditions each given 1000 simplex moves) and high sensitivity to initial conditions. This algorithm was carried out with 50,000 different initial conditions on a ten processor cluster and the best result of the 50,000 iterations was stored. Figure 7 shows the predictive behavior of randomized Nelder-Mead.

Figure 7 appears more predictive than any of the previous methods. The absolute sum of residuals was measured to be 995. The figure seems to be in the right range but not really predictive of nonlinear behavior. This algorithm may prove more useful when the time scale for experimental data is standardized.

2.1.3.5 Random Selection (Monte Carlo) Method

Although the random selection (RS) method is the second simplest algorithm, the computational time to get a high quality result is quite high. The algorithm employed in this method is rather simple. All parameters are scaled between 0 and 1 for rate constants and -1 and 1 for reaction orders. A 26-element column vector is randomly selected in the specified parameter range and then inserted into the nonlinear rate function. This process is repeated many times and the evaluation that gave the lowest sum absolute residuals was kept. When this algorithm was run 42 million times (computation time of roughly 12 hours on a single CPU) the lowest sum of absolute residuals was roughly 3,250. When the RS algorithm was run $500,000$ times (computational time of roughly 10 minutes) the minimum sum of absolute residuals was 9,300. The results for predictive nature of the model are seen

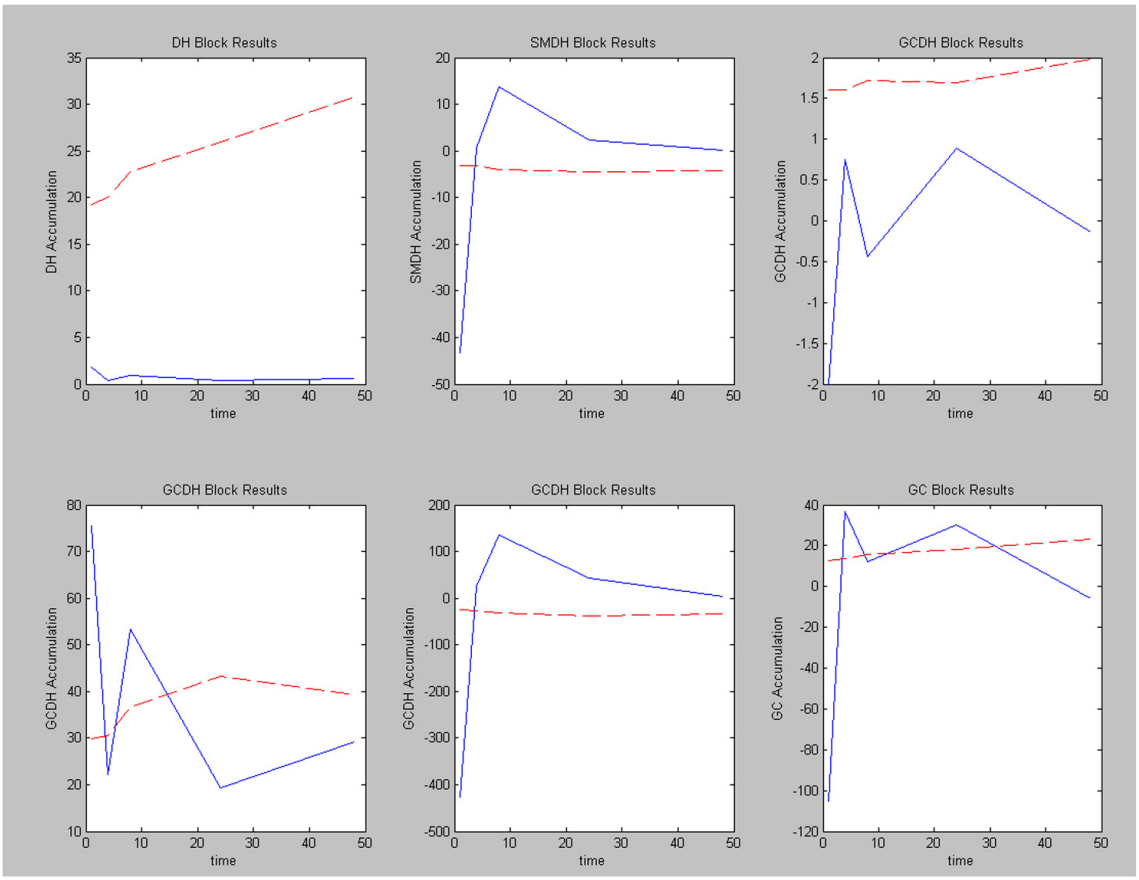

Figure 7: Randomized Nelder-Mead Fitting Results Phase 1: The solid blue line represents the derivative calculated from actual experimental data and the red dashed line represents the values predicted by the parameters from an NM nonlinear fitting.

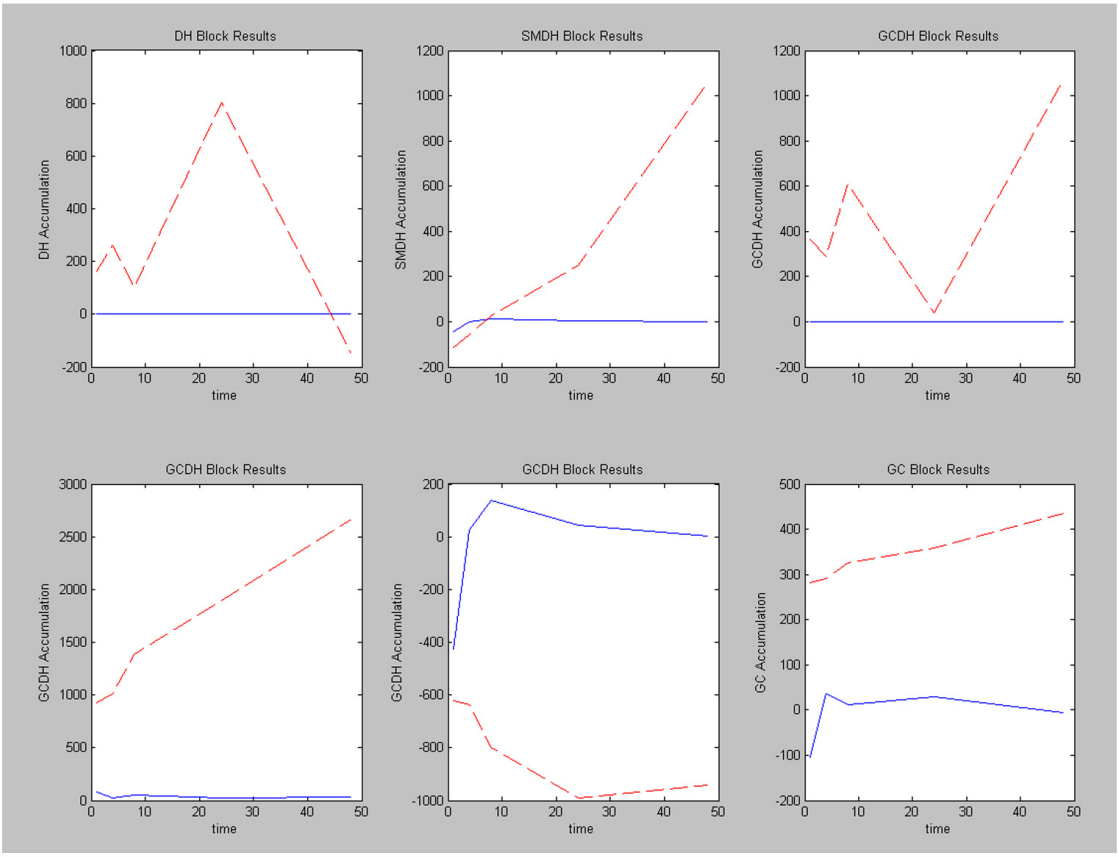

Figure 8: Random Selection (Monte Carlo) Method Results Phase 1: The solid blue line represents the derivative calculated from actual experimental data and the red dashed line represents the values predicted by the parameters from a RS nonlinear fitting.

in figure 8.

2.1.3.6 Generalized Simulated Annealing

The generalized simulated annealing algorithm is a Monte Carlo simulation technique that is physically analogous to the slow cooling of glass and other materials. The minimized cost function serves as the equivalent to the energy state and a control parameter represents the physical temperature. The technique has proven especially useful in certain np-hard problems where many local extrema surround the global minimum. Although GSA vastly outperforms direct search techniques in these types of problems, the added accuracy comes with a steep computational cost [68]. The GSA Matlab implementation, which was adapted from [41], is not only computational intense (a well tuned version takes roughly 10 minutes

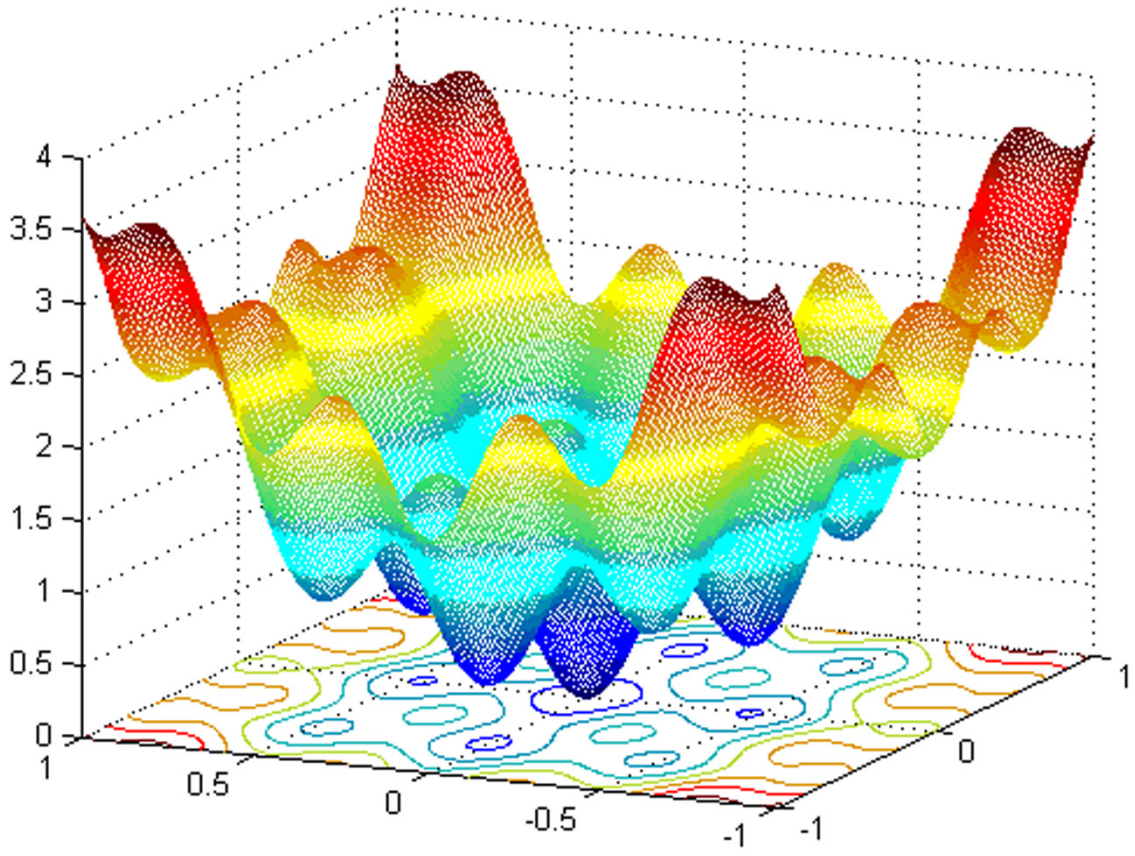

Figure 9: Cosmaze Simulated Annealing Test Function: The expectation landscape for the two dimensional cosmaze function has a global minimum at [0,0] and various local minima.

for a single iteration) but also highly sensitive to method parameters, both in terms of accuracy and time (a poorly tuned input could take several days to process, up to 1 million failed operations can be done in the inner while loop). The sensitivity to initial conditions was tested in a cosemaze function. The graph of this function is given in Figure 9.

Figure 9 demonstrates a two dimensional function in which most optimization functions will fail to find the global minimum unless the initial conditions are sufficiently close to the global minimum. The simulated annealing method consistently finds a minimum near the global minimum when adequate iterations are given. Figure 10 shows the path of the simulated annealing algorithm on cosmaze test function when starting at $[.85, .85]$ (blue) and $[-.5, -.75]$ (red). Each trial took approximately 4 seconds in real time to go through

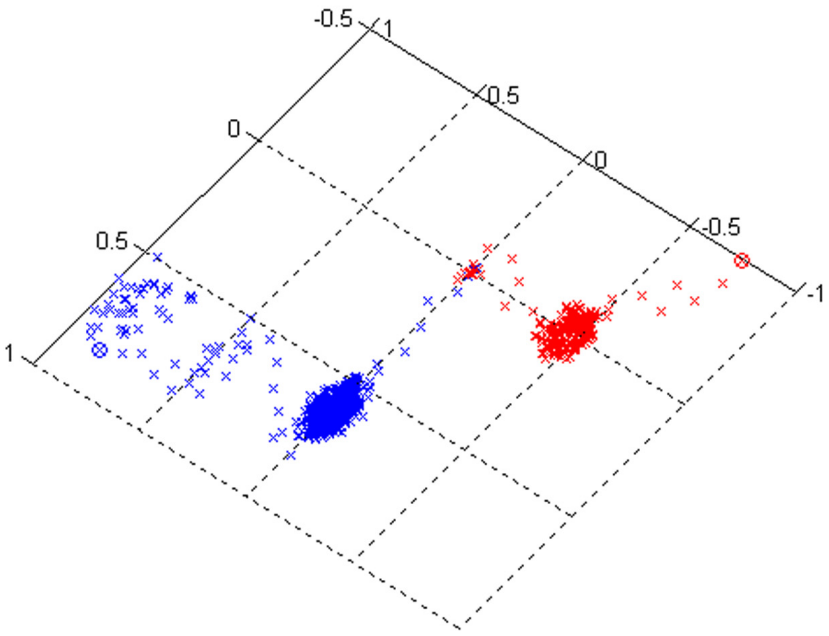

Figure 10: Annealing Progress to Global Minimum of Cosmaze Function: The iterations of the simulated annealing algorithm path on cosmaze test function are shown in the figure when starting at $[.85, .85]$ (blue) and $[-.5, -.75]$ (red).

$10,000$ algorithm iterations.

Because simulated annealing is a search method based on random movements, the method is not always consistent. Figure 10 shows that red trial reached the vicinity of the global minimum much faster than the blue trial but given the necessary iterations all methods would converge to the global minimum. The predictive behavior of a system using generalized simulated annealing fit parameters is compared to the original data in Figure 11. Figure 9 demonstrates a two dimensional function in which most optimization functions will fail to find the global minimum unless the initial conditions are sufficiently close to the global minimum. The simulated annealing method consistently finds a minimum near the global minimum when adequate iterations are given. Figure 10 shows the path of the simulated annealing algorithm on cosmaze test function when starting at $[.85, .85]$ (blue) and $[-.5, -.75]$ (red). Each trial took approximately 4 seconds in real time to go through $10,000$ algorithm iterations.

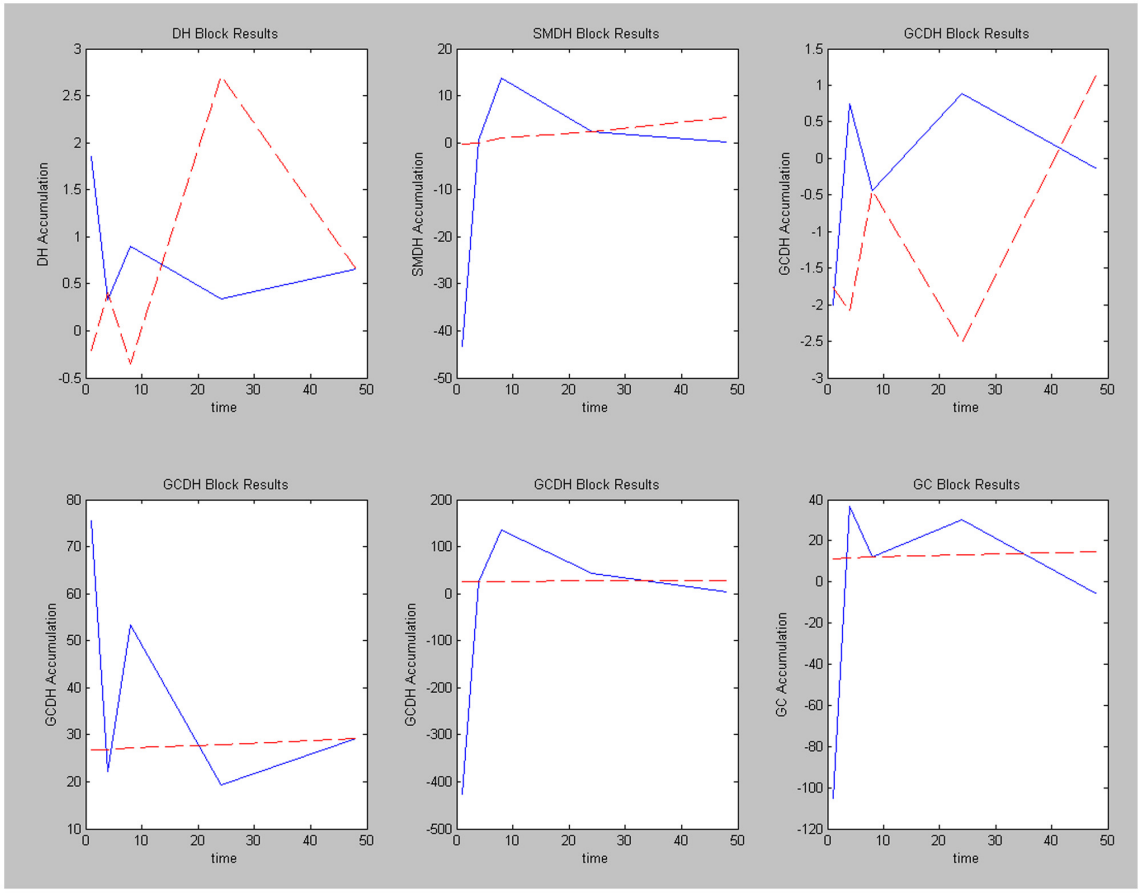

Figure 11: Generalized Simulated Annealing Results Phase 1: The solid blue line represents the derivative calculated from actual experimental data and the red dashed line represents the values predicted by the parameters from a GSA nonlinear fitting.

2.1.4 Critique and Conclusions

After the Phase 1 experiments were completed, several design aspects were deemed potential flaws, which prevented accurate characterization of the experimental data. First and foremost was the timing of the experimental measurements. These measurements were made at time points 0, 1, 4, 8, 24, 48 hours. When these inconsistent intervals were combined with the necessity to calculate an experimental derivative to fit against, a good amount of error may have been introduced especially since the state in which the cells were in at 48 hours was thought to be very different than at the earlier time points. The process was also very user intensive from the building of the matrices to represent the nonlinear rate function to the adjusting and manipulating the parameters of the various optimization methods especially simulated annealing. The hope is to automate a majority of these user intensive tasks in the future phases. Simulated annealing even with its additional computational cost appeared superior to other methods because of its ability to find a global minimum in even if passing through a global minimum was required. Although Phase 1 failed in accurately characterizing the experimental data, a good portion of the groundwork was laid and goals for future phases were determined.

2.2 General Algorithm Design Phase 2

The primary goals of the second phase of algorithm development were the following: further automating and enabling the automatic creation of the nonlinear rate function from a minimal set of text files, allowing the direct comparison of experimental measurements to predict values, building in a framework that allows algorithm progress tracking for cost, parameter values, and process time. Figure 12 shows the basic algorithm designs for the third stage of algorithm development.

Notice the addition of two elements in the Phase 2 design in comparison to the previous stage of development. The control script was added to provide algorithm flexibility by altering the script that drives the fitting routine from the command line instead of necessitating a change in the source code. An integrator module was also added to the scheme to allow the comparison of predicted values directly with the experimental observations. The

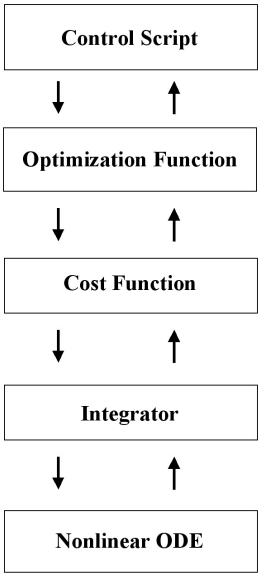

Figure 12: High Level Algorithm Design Phase 2: The relationship between the five primary components of the design are shown.

integrator module will be discussed at length in a later part of this section.

2.2.1 Building the Nonlinear Rate Function

As seen in the previous phase, the representation of the nonlinear rate function became increasingly more time demanding to produce as the number of nodes in the network increased. Not only was the computer code readability becoming more of an apparent problem but also the inflexibility of previous rate functions was clearly in opposition to the long-term project goals. Although capable of being mathematically identical to the both of the previous rate equations, the phase three implementation of the nonlinear rate function allowed flexibility in that any size network could be represented by simply redefining the R and F matrices. The F matrix or "footprint" is a system of -1s, 0s, and 1s that describe how the terms in the equation relate to a particular species, which is indicated by the row of the matrix. The following matrix in equation 94 represents the F matrix for the previously

discussed six-node sphingolipid system.

$$F = \begin{bmatrix} -1 & 1 & -1 & 1 & -1 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \tag{94}$$

Each of the ten columns represents one term in the composite rate equation and each row represents a particular node. Using the above matrix, the first nodes rate equation is comprised of a negative term 1, a positive term 2, a negative term 3, a positive term 4, a negative term 5, and a positive term 6; whereas, the second node is comprised of a positive term 1 and a negative term 2.

The "footprint" matrix is not sufficient by itself to represent the nonlinear rate term a second matrix called the R or "reaction" matrix is required to describe the type of network connectivity. The R matrix is the same size as the F matrix and is composed of 1s and 0s. Each pair of columns represents a single reversible chemical reaction. The first of the pair represents the presence of reverse reaction in the rate equation while the second represents the presence of the forward reaction in the rate equation. The R matrix for the six-node system is shown in equation 95.

$$R = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{95}$$

The previous matrix effectively states that the first node rate equation is composed of three reverse reactions, one with node 2, one with node 3, and the final one with node 4. The fourth node's rate equation is made from a forward reaction with node 1, a reverse reaction with node 5, and a reverse reaction with node 6.

2.2.2 Integrator Module

The integration of the rate function system can be treated as initial value problem thus allowing the direct comparison of experimental observations and predicted values. This comparison is achieved by numerically integrating the system of nonlinear rate functions to produce a predicted concentration instead of a concentration rate of change as found in the initial two phases. Adaptive step size Runge-Kutta was selected as the numerical integration method because of its ability to provide efficient highly accurate solutions, its simplicity and ease to implement, and its wide spread use [72].

Fehlberg proposed the original adaptive step size Runge-Kutta algorithm. The Fehlberg algorithm consists of embedded Runge-Kutta formulae of order four and five in which the fifth order formula is used as an error metric to determine the accuracy of the fourth order solution. If the terms agree within a predefined tolerance, the step is accepted and the step size for the next step is enlarged proportional to the degree by which the tolerance was exceeded. If the difference in the terms is beyond the preset tolerance, the step is rejected and the step repeated with the step size is reduced to a degree, which the algorithm predicts will meet the tolerance. The following equations, 96 – 104, constitute the Runge-Kutta-Felhberg Method [53].

$$k_1 = hf(t_k, y_k) \tag{96}$$

$$k_2 = hf(t_k + \frac{1}{4}h, y_k + \frac{1}{4}k_1) \tag{97}$$

$$k_3 = hf(t_k + \frac{3}{8}h, y_k + \frac{3}{32}k_1 + \frac{9}{32}k_2) \tag{98}$$

$$k_4 = hf(t_k + \frac{12}{13}h, y_k + \frac{1932}{2197}k_1 - \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3) \tag{99}$$

$$k_5 = hf(t_k + h, y_k + \frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4) \tag{100}$$

$$k_6 = hf(t_k + \frac{1}{2}h, y_k - \frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5) \tag{101}$$

The previous equations are used to produce the fourth and fifth order approximations given by the following equations [53]. y represents the fourth order approximation and z

represents the fifth order approximation.

$$y_{k+1} = y_k + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5 \tag{102}$$

$$z_{k+1} = y_k + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28,561}{56,430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6 \tag{103}$$

The calculation of step size scalar s, which controls the amount by which the step size increases or decreases, is given by equation 104 below.

$$s = \left(\frac{tol * h}{2|z_{k+1} - y_{k+1}|}\right)^{1/4} \tag{104}$$

Several advances in adaptive step size Runge-Kutta have been made since Fehlberg's original formulation. One sight alteration is to use the fourth order formula as error metric for the fifth order approximation. Another improvement is to use the slightly more efficient constants derived by Cash and Karp [14]. These alterations and a practical step size adjustment suggested by Press et al. were incorporated to produce a suitable integration function [72].

2.2.3 Cost Function

The Phase 2 cost function is identical to the Phase 1 cost function except the Phase 2 cost function incorporated the actual observations instead of the derivatives. The data was input as a simple matrix where the first column is the time points and the subsequent columns are the concentrations of the nodes at the given time points. For example, the six-node system discussed earlier with six time points would give rise to a 6x7 matrix. The equations that were used to provide the data fit are given below in equations 105 – 110. The implementation of this cost function was also designed to provide a high cost for parameters that are overly difficult to integrate.

$$C_{DH} = \int_0^6 M_{DH} - k_1 C_{DH}^{\alpha_1} + k_2 C_C^{\alpha_2} - k_3 C_{DH}^{\alpha_3} + k_4 C_{SMDH}^{\alpha_4} - k_5 C_{DH}^{\alpha_5} + k_6 C_{GCDH}^{\alpha_6} dt \tag{105}$$

$$C_{SMDH} = \int_0^6 M_{SMDH} + k_3 C_{DH}^{\alpha_3} - k_4 C_{SMDH}^{\alpha_4} dt \tag{106}$$

$$C_{GCDH} dt = \int_0^6 M_{GCDH} + k_5 C_{DH}^{\alpha_5} - k_6 C_{GCDH}^{\alpha_6} dt \tag{107}$$

$$C_C = \int_0^6 M_C + k_1 C_{DH}^{\alpha_1} - k_2 C_C^{\alpha_2} - k_7 C_C^{\alpha_7} + k_8 C_{SM}^{\alpha_8} - k_9 C_C^{\alpha_9} + k_{10} C_{GC}^{\alpha_{10}} dt \tag{108}$$

$$C_{SM} = \int_0^6 M_{SM} + k_7 C_C^{\alpha_7} - k_8 C_{SM}^{\alpha_8} dt \tag{109}$$

$$C_{GC} = \int_0^6 M_{GC} + k_9 C_C^{\alpha_9} - k_{10} C_{GC}^{\alpha_{10}} dt \tag{110}$$

The cost function and optimization method are intertwined in the cases where nonlinear regression was used to fit the data. In the intertwined case the cost function is the least residual squares. In cases where the optimization method and cost function are separate the fitting method is the sum of absolute residuals. The method of calculating cost in instances where the optimization method and cost function are separate is given by equations 111 – 114.

$$Cost1 = |C_{DH}^{DATA} - C_{DH}^{CALC}| + |C_{SMDH}^{DATA} - C_{SMDH}^{CALC}| \tag{111}$$

$$Cost2 = |C_{GCDH}^{DATA} - C_{GCDH}^{CALC}| + |C_C^{DATA} - C_C^{CALC}| \tag{112}$$

$$Cost3 = |C_{SM}^{DATA} - C_{SM}^{CALC}| + |C_{GC}^{DATA} - C_{GC}^{CALC}| \tag{113}$$

$$Cost = Cost1 + Cost2 + Cost3 \tag{114}$$

The above equations show the experimental measurement of each species being compared to the calculated measurement of each species. The absolute values of these comparisons are summed to give the cost, which is reported to the optimization function.

2.2.4 Critique and Conclusions

Phase 2 was a success in that the code required for fitting not only was universal for any network but also was substantially compacted and more readable. Other advantages include the ability to track the progress of the algorithm as it accepts steps, and its compatibility to run on both on ia32 and ia64 systems.

The major disadvantage was the system of equations was sufficiently stiff; meaning the dependent variables (the node concentrations) changed on very different independent variable scales (the kinetic parameters). Most chemical kinetics problems are sufficiently stiff and require a numerical integration method that remains stable [42]. The implemented explicit Runge-Kutta integrator was unable to integrate the system of rate equations without taking extremely minute step sizes, which often times exceeded the limits of floating point mathematical operations.

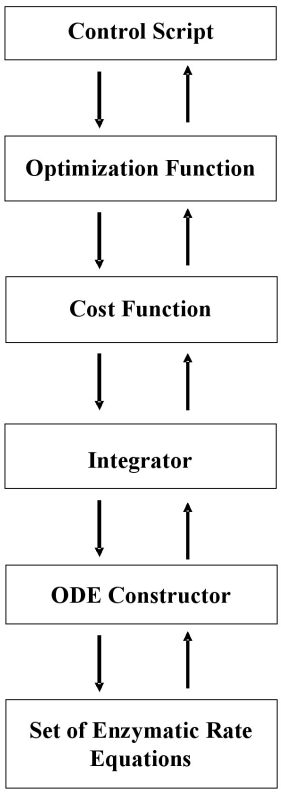

Figure 13: High Level Algorithm Design Phase 3: The relationship between the six primary components of the design are shown.

2.3 General Algorithm Design Phase 3

The primary goals of the third design phase were to incorporate a stiff integrator and provide an interchangeable set of enzymatic reaction models. The design map of Phase 3 is given in Figure 13. Notice in Figure 13 that the insertion of two pieces, the ODE Constructor, and the set of Enzymatic Rate Equations, have replaced the nonlinear rate function, which was used in all previous phases.

2.3.1 Enzymatic Rate Equation

Since experimental data is only available for a single substrate and single product for each of the numbered reactions in the network map in Figure 14, only the Uni Uni Mechanism was used in this phase. The process of adding more enzymatic rate equations will be described in the section about the ode constructor. The velocity for the five Uni Uni reactions (numbered

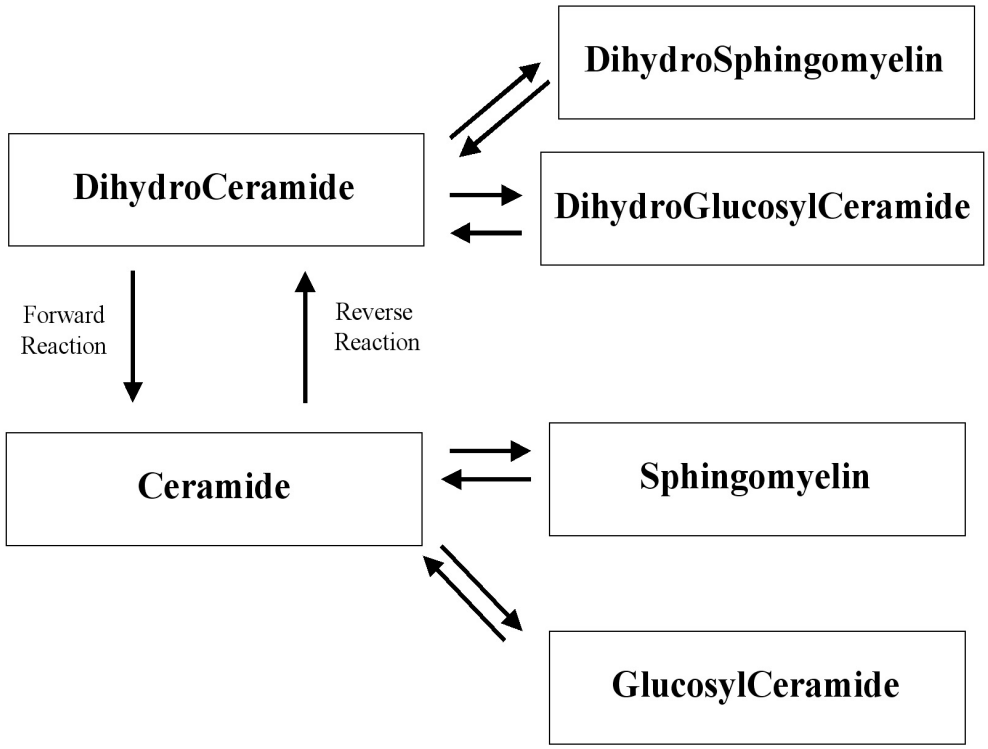

Figure 14: Six-Node Network Diagram: Each of the Enzymatic Reaction are numbered to reflect the labeling through out Phase 3.

in Figure 14) used in this phase are given in equations 115 – 119.

$$V_1 = \frac{k_1k_3k_5[C_{DH}][E_1] - k_2k_4k_6[C_{SMDH}][E_1]}{k_2k_5 + k_2k_4 + k_3k_5 + (k_1k_3 + k_1k_4 + k_1k_5)[C_{DH}] + (k_2k_6 + k_3k_6 + k_4k_6)[C_{SMDH}]} \tag{115}$$

$$V_2 = \frac{k_7k_9k_{11}[C_{DH}][E_2] - k_8k_{10}k_{12}[C_{GCDH}][E_2]}{k_8k_{11} + k_8k_{10} + k_9k_{11} + (k_7k_9 + k_7k_{10} + k_7k_{11})[C_{DH}] + (k_8k_{12} + k_9k_{12} + k_{10}k_{12})[C_{GCDH}]} \tag{116}$$

$$V_3 = \frac{k_{13}k_{15}k_{17}[C_{DH}][E_3] - k_{14}k_{16}k_{18}[C_C][E_3]}{k_{14}k_{17} + k_{14}k_{16} + k_{15}k_{17} + (k_{13}k_{15} + k_{13}k_{16} + k_{13}k_{17})[C_{DH}] + (k_{14}k_{18} + k_{15}k_{18} + k_{16}k_{18})[C_C]} \tag{117}$$

$$V_4 = \frac{k_{19}k_{21}k_{23}[C_C][E_4] - k_{20}k_{22}k_{24}[C_{SM}][E_4]}{k_{20}k_{23} + k_{20}k_{22} + k_{21}k_{23} + (k_{19}k_{21} + k_{19}k_{22} + k_{19}k_{23})[C_C] + (k_{20}k_{24} + k_{21}k_{24} + k_{22}k_{24})[C_{SM}]} \tag{118}$$

$$V_5 = \frac{k_{25}k_{27}k_{29}[C_C][E_5] - k_{26}k_{28}k_{30}[C_{GC}][E_5]}{k_{26}k_{29} + k_{26}k_{28} + k_{27}k_{29} + (k_{25}k_{27} + k_{25}k_{28} + k_{25}k_{29})[C_C] + (k_{26}k_{30} + k_{27}k_{30} + k_{28}k_{30})[C_{GC}]} \tag{119}$$

2.3.2 ODE Constructor

The purpose of the ODE constructor is to build the governing ordinary differential equation for each node from a specified combination of enzymatic equations. The "footprint" matrix, is made of -1 's, 0 's, and 1 's, gives the combination of required equations for each node. The "footprint" matrix of the six-node system is given in equation 120. Each row describes the rate equations that a particular species is involved. A -1 means the species of that row is the substrate in the particular reaction given by the column. For example, species 1 (dihydroceramide) is the substrate in reaction 1, 2, and 3, whereas, species 2 (dihydrosphingomyelin) is the product in the first reaction.

$$F = \begin{bmatrix} -1 & -1 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{120}$$

The R matrix has been simplified from the previous versions. The R, "reaction", matrix is now a row vector with length number of reactions which describes what type of enzymatic reaction is required for each reversible reaction in the network. Since only the Uni Uni Mechanism was available due to the experimental constraints, the six-node R matrix is a vector of ones length five, which is given in equation 121.

$$R = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{121}$$

Once the enzymatic rate functions are calculated, the system of ordinary differential equations to be integrated is given by the expression in equation 122.

$$deriv = (\sum (F * diag(V))^T)^T \tag{122}$$

2.3.3 Integrator

The integrator was in this third phase was the odes15s integrator from the Matlab ODE Suite [85]. This integrator is valid for both stiff and nonstiff sets of ordinary differiential equation initial value problems. This integrator uses either Backward Differentiation Formulas (BDFs), as known as Gears Methods, and Numerical Differentiation Formulas (NDFs) [85].

2.3.4 Cost Function

The cost function in the third stage is again a comparison of the experimental observations and a predicted concentration derived from the integration of the composite rate function. The equations for the third phase are given below in equations 123 – 126.

$$Cost1 = |C_{DH}^{DATA} - C_{DH}^{CALC}| + |C_{SMDH}^{DATA} - C_{SMDH}^{CALC}| \tag{123}$$

$$Cost2 = |C_{GCDH}^{DATA} - C_{GCDH}^{CALC}| + |C_{C}^{DATA} - C_{C}^{CALC}| \tag{124}$$

$$Cost3 = |C_{SM}^{DATA} - C_{SM}^{CALC}| + |C_{GC}^{DATA} - C_{GC}^{CALC}| \tag{125}$$

$$Cost = Cost1 + Cost2 + Cost3 \tag{126}$$

2.3.5 Results and Discussion

The Phase 3 design unlike Phase 2 design was able to produce results. The experimental measurement of a particular species concentration is shown in the plots below as a blue circle, whereas the predict value of the rate is seen as a dashed red line.

A quick glance at Figure 15 shows that the kinetic parameters found by this fitting method are not ideal. Figure 15 was from the best-fit trial in terms of cost, ending cost value of roughly 438. To put this into perspective, if all zeros were selected as the kinetic parameters, the resulting cost is roughly 500. One possible explanation of the poor fit is that the algorithm was simply not run long enough. Figure 16 shows the progression of the algorithm in terms of cost as computational time increases.

Figure 16 clearly shows nearly flat line improvement from the simulated annealing algorithm after roughly 800 seconds. The algorithm was clearly run long enough in this case

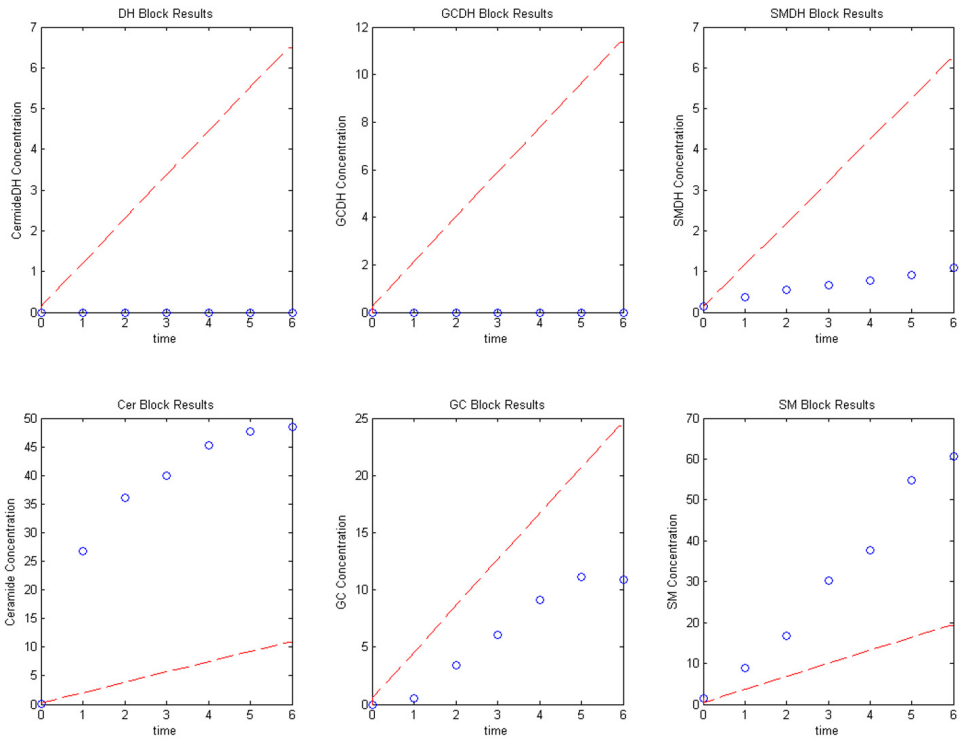

Figure 15: Generalized Simulated Annealing Results Phase 3: The blue circles represent the derivative calculated from actual experimental data and the red dashed line represents the values predicted by the parameters from a GSA nonlinear fitting.

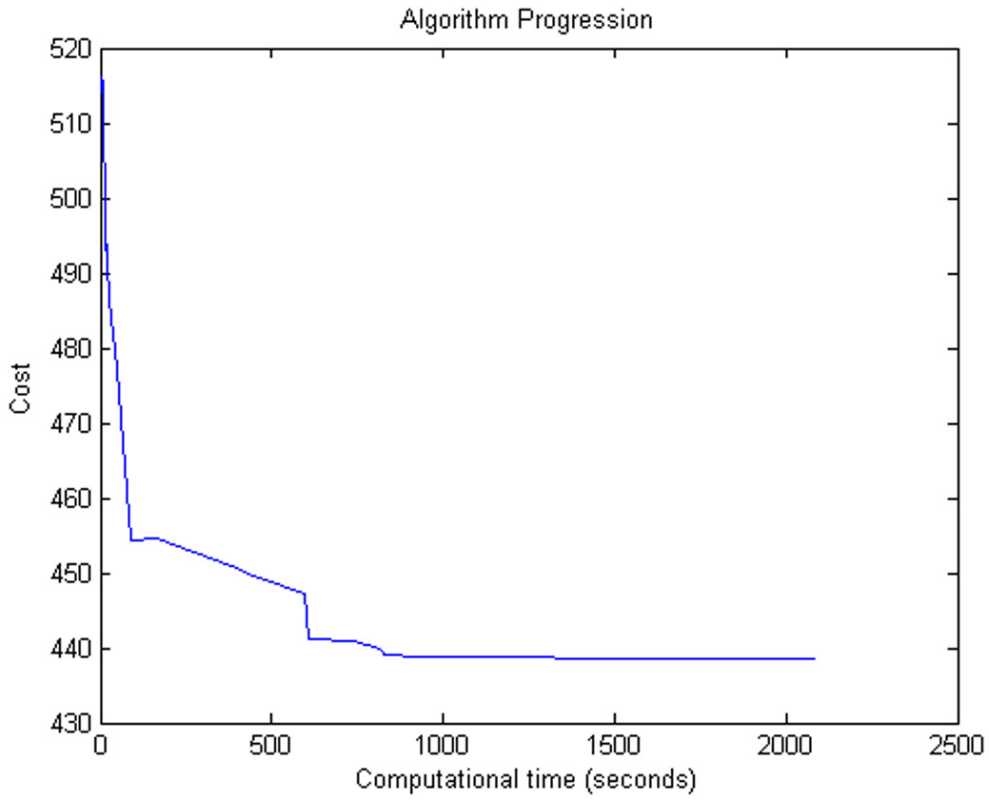

Figure 16: Generalized Simulated Annealing Results Phase 3 Algorithm Progression: The plot shows the progression of the simulated annealing method for the best fit trial against time.

barring a sudden drop in cost after a period of flat lining. Another potential explanation of the algorithms lackluster performance is that the algorithm got stuck in a neighborhood that it could not leave. After analyzing the conditions under which the simulated annealing procedure operated, it is rather easy to see how the algorithm could stick in a certain area. The kinetic parameters, which have a potential dynamic range or ten orders of magnitude, were scaled from 0 to 1 then multiplied by 10^9 to apply the correct range. Simulated Annealing, as mentioned before, operates using a uniform random number generator. If random initial conditions are selected, then on average 99% of the kinetic parameters will lie in the region between 10^7 and 10^8. This is clearly not what was intended and locks the algorithm in this neighborhood of a single order of magnitude. Several methods that attempted to distribute the initial conditions were devised including using a logarithm. These methods potentially allowed the system of equations to become too stiff for the integrator in nearly 1 in 1,000 cases. If 1,000 random guesses were made consecutively, the integrator showed a failure rate of roughly 66% in terms of being able to complete 1,000 successive iterations.

2.3.6 Critique and Conclusions

Although algorithm development has gained substantially on the overall goal through the completion of the three phases, a quality, accurate, relevant solution has still not been elucidated. The failure of a commercial integrator was clearly not foreseen but not totally unexpected since this fitting problem has been actively pursued for some time without success. The next logical steps are first to produce an integrator or other function capable of distinguishing ultra-stiff problems, which are not likely to be quality results. Another worthy task is to study and facilitate optimal fitting performance. This task requires extensively studying the paths and preferences of optimization and fitting methods.

CHAPTER III

VALIDATION OF THE MODELING FRAMEWORK WITH SIMULATED DATA

The purpose of the work in this chapter is to elucidate the kinetic parameters that govern a simulated sphingolipid metabolism system using various global optimization routines including Monte Carlo, Simulated Annealing, and Genetic Algorithms. Here a simulated six node system was built from five UniUni reaction equations with known kinetic parameters. Each node was treated as a combination of single substrate, single product catalyzed reactions. This defined system of equations is then sampled at a rate that mimics the mass spectrometry measurements of the complex pathway in time shown in [34]. As the investigation on mathematical models of biological events continues to gain popularity, the use of global optimization methods to quickly and reliably estimate missing parameters will become more vital. This work investigates the use of global parameter estimation schemes in terms of their reliability to the true underlying kinetic parameters. When the amount of fitting parameters is sufficiently large, it is likely to find parameter sets that predict the data decently well, but the optimized parameters often are not closely related to the true underlying parameters.

3.1 Methods

In order to meet the objectives of the study, several computational tasks had to be pursued. The first of these tasks was setting a suitable simulated network. The other critical task in this study was to prepare implementations of the necessary optimization methods.

3.1.1 Simulated System Design

A simulated six-node system was developed to mirror the topology of the network in Figure 17. Each of the reversible connections, shown in the figure as two arrows, is modeled as a

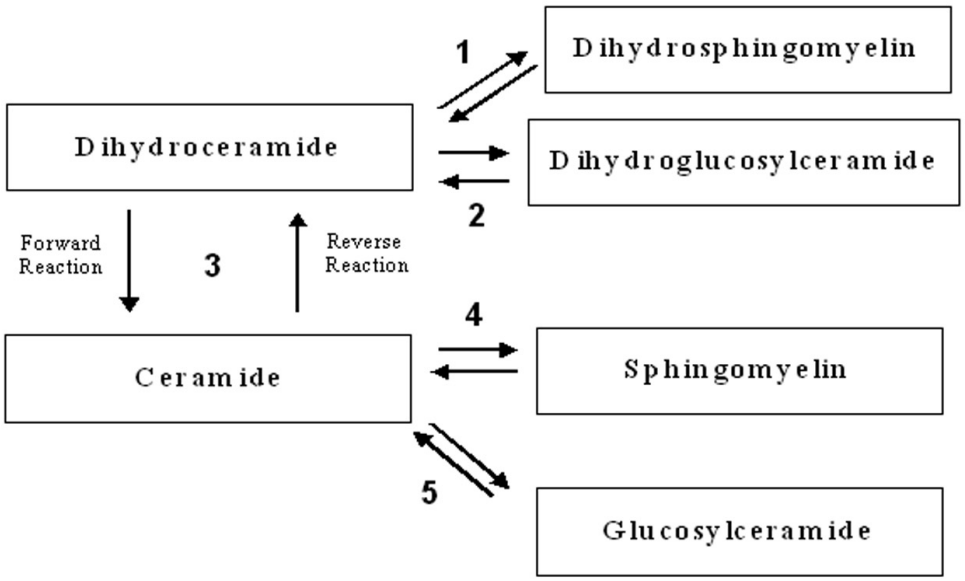

Figure 17: Six-Node Network with Numbered Enzymatic Reactions: The species listed here are real metabolites in the sphingolipid metabolism pathway. This network is only a small section of the larger sphingolipid and metabolism networks (See www.sphingomap.org for complete pathway). Typically, this network is highly influenced by surrounding pathways but here we consider it as an isolated system.

kinetic UniUni mechanism [51, 44].

3.1.2 Monte Carlo Implementation

Monte Carlo methods have been applied in many problems, most notably integration. The Monte Carlo methods employed in this work consist of picking a vector of random numbers that represent the kinetic rate constants. A uniform distribution from zero to one is returned for each element of the vector. This number between zero and one directly maps to a rate constant between 0.1 and 10^9. Care was taken to insure that roughly an equal probability for each order of magnitude was provided.

3.1.3 Simulated Annealing Implementation

A simulated annealing methodology was adapted from a Matlab routine used to optimize parameters in single chemical reactions [41]. In addition to increasing the number of

dimensions being optimized, several tests were run to investigate if the parameters of the algorithm needed to be adjusted due to the scale change. This optimization method also operates on scaled parameters with a range between 0 and 1. This number between zero and one directly maps to a rate constant between 0.1 and 10^9.

3.1.4 Genetic Algorithm Implementation

First, an initial population of 50 individuals, each with 30 genes representing the 30 parameters being fit, is randomly generated. The cost of all of the individuals is evaluated in a systematic fashion. Arbitrarily high cost values are given to the individuals with genes out of 0 to 1 range and those that are unable to be integrated. After cost values are assigned to each of the population members, the two members with the lowest cost are placed in the next generation. The process of filling the remaining 48 spots in the next generation starts with selecting parents from the current generation. A selection procedure was used to insure that the more fit individuals had a higher probability of having their genes represented in the next generation. After a suitable number of parents have been selected, 95% of the next generation is formed through crossover of two parents and the remaining 5% is done randomly mutating the parents a random number of times. Now a new generation is formed and process iterates in order to produce the following generation and continues until a defined number of generations have passed.

3.2 Results and Discussion

The results of this work are classified in two different ways. First the computational kinetic parameters are judged according to how well they describe the system behavior. In order to show this behavior fitness, the synthetic data points (red circles), which were used to get the cost in each of the optimization routines, were plotted against a line that describes the time course behavior for the particular metabolite. These graphs were prepared for each node in the network. Figure 18 shows the behavior fitness for the lowest cost result in the first Monte Carlo trial.

Plots showing the best-fit results for the second Monte Carlo trial are shown in Figure 19. Again, here the red circles represent the data points being fit to and the blue line

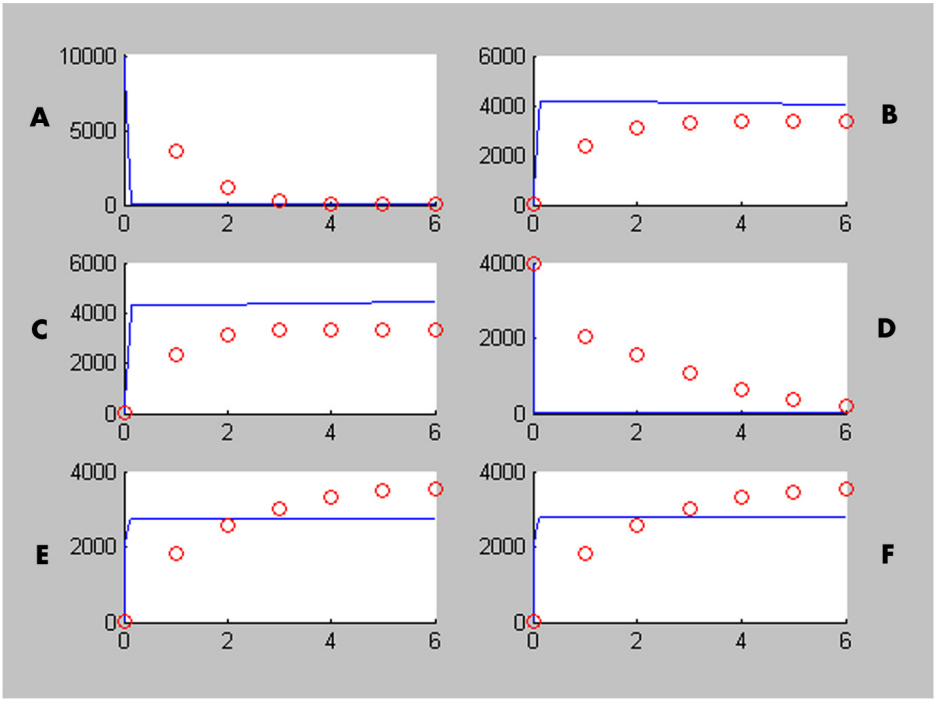

Figure 18: Best-Fit Monte Carlo Trial 1: All plots represent the concentration per unit cell number of a particular metabolite species in time. The circles represent the data measurement in which the random trials of the Monte Carlo study were compared. All units on the y-axis are pico-moles per unit cell number. The units for time on the x-axis are hours. Together the plots describe a hypothetical six node five connection metabolism network analogous to the one described in the system design section. Plot A represents the hypothetical accumulation of dihydroceramide. Plot B represents the hypothetical accumulation of dihydroglucoslyceramide. Plot C is the accumulation of dihydrosphingomyelin. Plot D is the ceramide accumulation. Plot E is the glucoslyceramide accumulation. Plot F is the sphingomyelin accumulation.

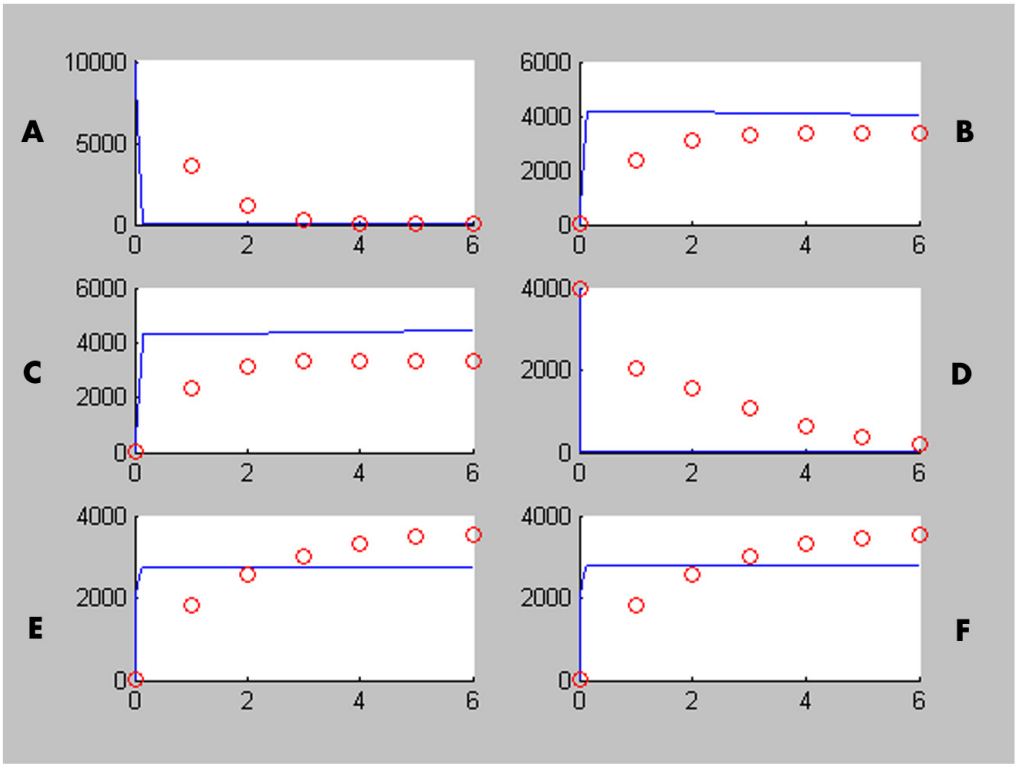

Figure 19: Best-Fit Monte Carlo Trial 2: All plots represent the concentration per unit cell number of a particular metabolite species in time. The circles represent the data measurement in which the random trials of the Monte Carlo study were compared. All units on the y-axis are pico-moles per unit cell number. The units for time on the x-axis are hours. Together the plots describe a hypothetical six node five connection metabolism network analogous to the one described in the system design section. Plot A represents the hypothetical accumulation of dihydroceramide. Plot B represents the hypothetical accumulation of dihydroglucoslyceramide. Plot C is the accumulation of dihydrosphingomyelin. Plot D is the ceramide accumulation. Plot E is the glucoslyceramide accumulation. Plot F is the sphingomyelin accumulation.

represents the time course of each metabolite as predicted by the best kinetic parameters.

Plots showing the best-fit results for the first Simulated Annealing trial are shown in Figure 20. Again, here the red circles represent the data points being fit to and the blue line represents the time course of each metabolite as predicted by the best kinetic parameters.

Plots showing the best-fit results for the second Simulated Annealing trial are shown in Figure 21. Again, here the red circles represent the data points being fit to and the blue line represents the time course of each metabolite as predicted by the best kinetic parameters.

Plots showing the best-fit results for the Genetic Algorithm trial are shown in Figure 22.

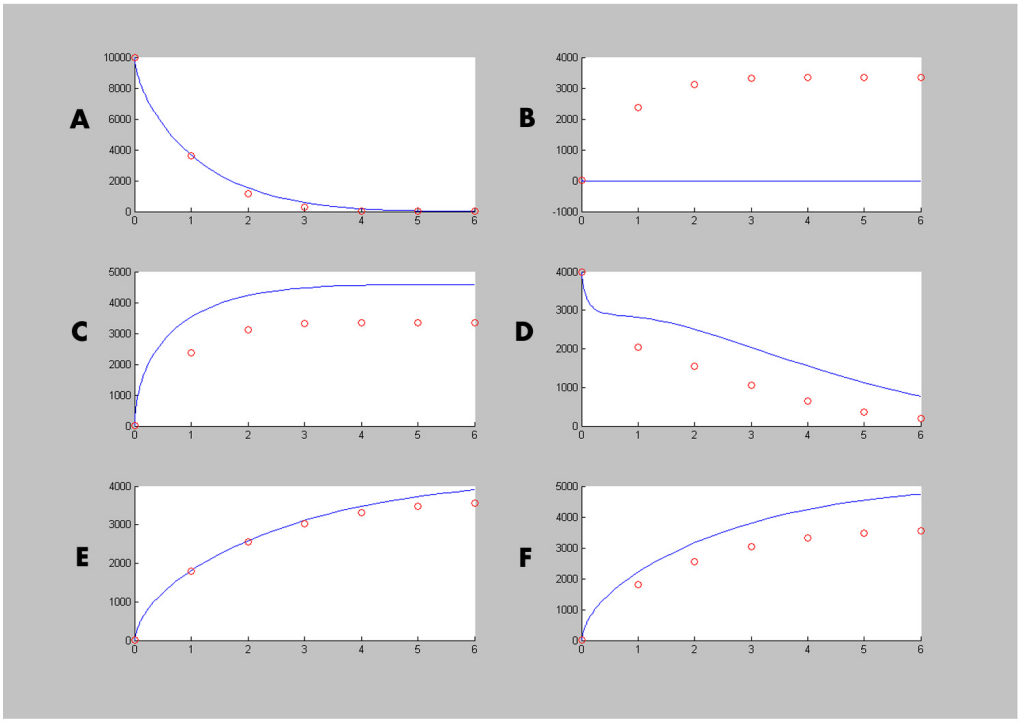

Figure 20: Best-Fit Simulated Annealing Trial 1: All plots represent the concentration per unit cell number of a particular metabolite species in time. The circles represent the simulated data measurements the algorithm is fitting (runtime of 5.5 hours). All units on the y-axis are pico-molar per unit cell number. The time units on the x-axis are hours. Together the plots describe a hypothetical six node five connection metabolism network analogous to the one described in the system design section. Plot A represents the hypothetical accumulation of dihydroceramide. Plot B represents the hypothetical accumulation of dihydroglucoslyceramide. Plot C is the accumulation of dihydrosphingomyelin. Plot D is the ceramide accumulation. Plot E is the glucoslyceramide accumulation. Plot F is the sphingomyelin accumulation.

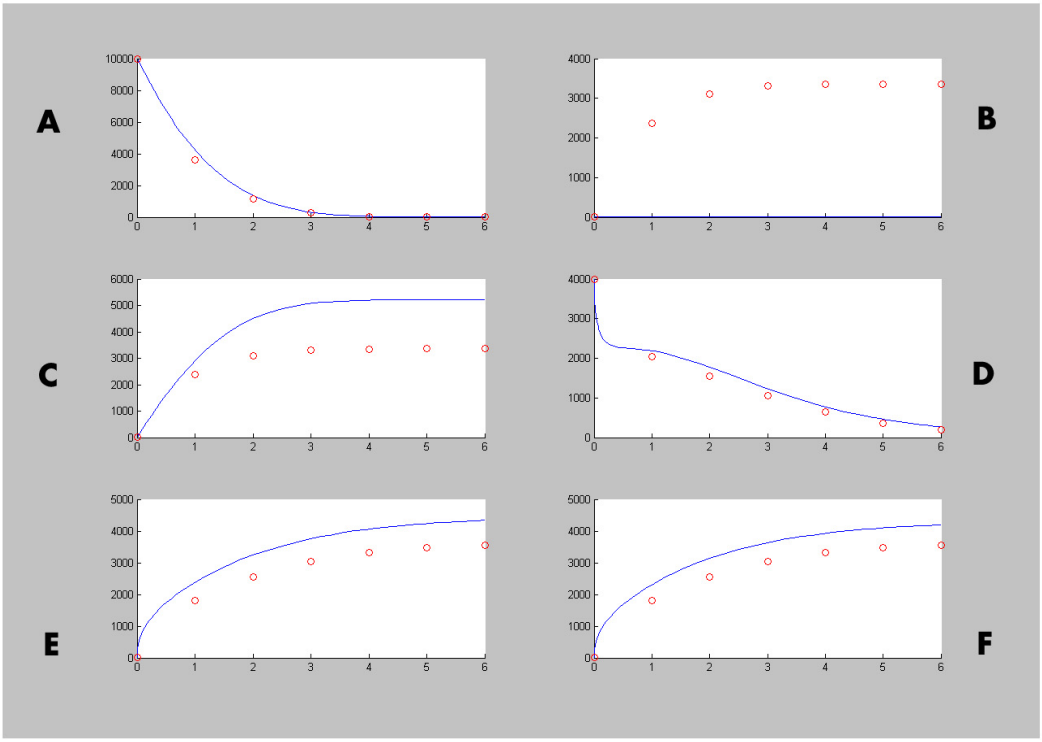

Figure 21: Best-Fit Simulated Annealing Trial 2: All plots represent the concentration per unit cell number of a particular metabolite species in time. The circles represent the simulated data measurements the algorithm is fitting (runtime of 5.5 hours). All units on the y-axis are pico-molar per unit cell number. The time units on the x-axis are hours. Together the plots describe a hypothetical six node five connection metabolism network analogous to the one described in the system design section. Plot A represents the hypothetical accumulation of dihydroceramide. Plot B represents the hypothetical accumulation of dihydroglucoslyceramide. Plot C is the accumulation of dihydrosphingomyelin. Plot D is the ceramide accumulation. Plot E is the glucoslyceramide accumulation. Plot F is the sphingomyelin accumulation.

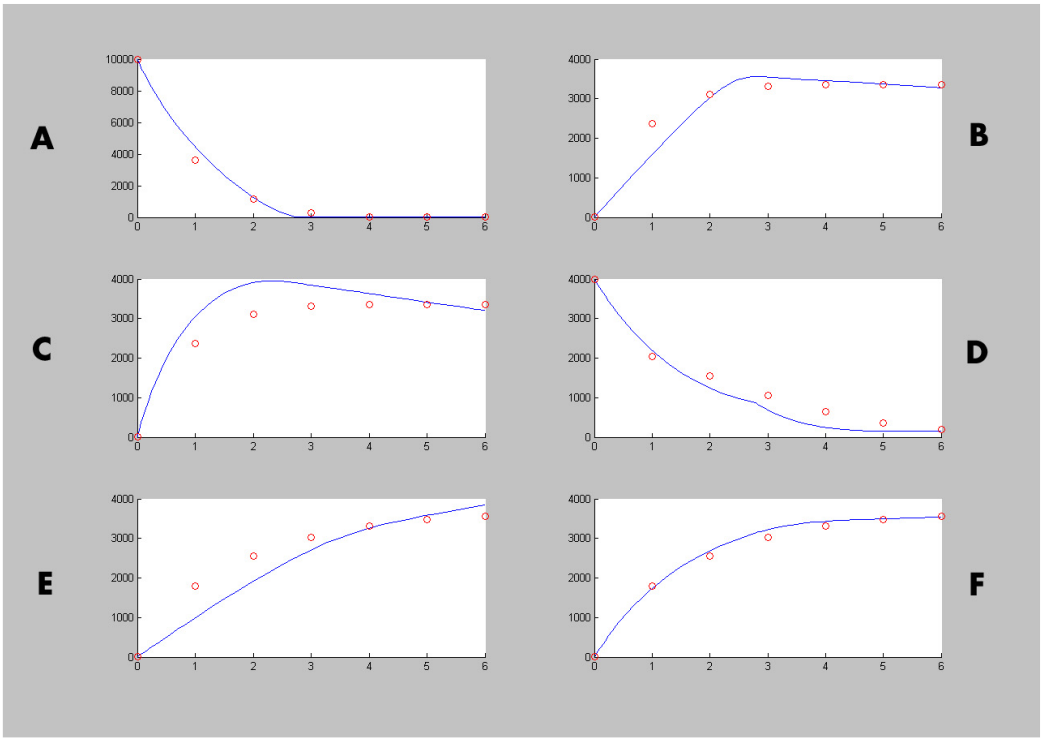

Figure 22: Best-Fit Genetic Algorithm: All plots are the concentration per unit cell number of a particular metabolite species in time. The circles represent the simulated data measurements the algorithm is fitting. All units on the y-axis are pico-moles per unit cell number. The time units on the x-axis are in hours. Together the plots describe a hypothetical six node five connection metabolism network analogous to the one described in the system developement section. Plot A represents the hypothetical accumulation of dihydroceramide. Plot B represents the hypothetical accumulation of dihydroglucoslyceramide. Plot C is the accumulation of dihydrosphingomyelin. Plot D is the ceramide accumulation. Plot E is the glucoslyceramide concentration versus time. Plot F is the sphingomyelin accumulation.

Again, here the red circles represent the data points being fit to and the blue line represents the time course of each metabolite as predicted by the best kinetic parameters.

The computational output can be investigated not only by examining overall network behavior but also by comparing the predicted kinetic parameters to the known values since the network was built from known parameters. Table 1 shows the true kinetic parameters and the kinetic parameters that were used to generate the previous figures.

In contrast to what we see in figures 19-22, Table 1 shows that the actual kinetic parameters used to construct the fitted line are significantly different from the true underlying parameters. The measure of how well the fitted line fits to the data also does not seem to

Table 1: True Kinetic Parameters from Simulated System (TRUE) and Best Fit Parameters for 2 Monte Carlo Trials (MC1 and MC2), 2 Simulated Annealing Trials (SA1 and SA2), and a Genetic Algorithm trial (GA).

Parameter/ Method	TRUE	MC1	MC2	SA1	SA2	GA
k1	130	5.98E+05	6.78E+05	10.345	75509	7.76E+08
k2	110	3.91E+07	0.031022	3.43E+08	1.07E+07	1.09E+05
k3	5.00E+06	3.28E+07	1.02E+07	681.03	14352	9.19E+06
k4	8.00E+05	47746	26744	1.16E+07	11.033	4.31E+07
k5	500	3.794	5.22E+07	1.03E+08	9.59E+08	7.0525
k6	495	40381	90928	1.07E+08	3.57E+06	9.54E+07
k7	130	2.19E+06	1.76E+07	7.34E+07	1.76E+05	25055
k8	110	961.75	1.31E+05	441.41	4.69E+09	6.63E+06
k9	5.00E+06	6.90E+07	1.75E+06	7.42E+07	7.62E+06	8.71E+07
k10	8.00E+05	61.337	14329	76057	7679.2	3.98E+06
k11	500	1.00E+08	35006	70.02	2.47E+06	9.07E+08
k12	495	5.4015	6646.2	4.89E+09	1.5466	9.69E+05
k13	130	95711	8.40E+08	9.97E+07	1.03E+06	5.04E+06
k14	110	7.56E+06	8.24E+09	8.51E+06	1.04E+05	7.00E+06
k15	5.00E+06	1.20E+09	726.81	2.04E+07	9.13E+08	9.17E+05
k16	8.00E+05	1.07E+09	8.0722	130.66	997.01	0.8067
k17	500	7.67E+09	7.93E+09	76748	8.39E+08	0.6386
k18	495	4.10E+08	2.35E+07	8.46E+08	9.38E+08	10.677
k19	130	6.66E+06	8675.7	5.24E+07	4.6415	5.94E+07
k20	110	4.07E+06	2366.6	99263	73.209	6.31E+09
k21	5.00E+06	6.02E+08	9.00E+07	2.70E+07	7.69E+09	1.01E+06
k22	8.00E+05	859.98	191.88	1033.6	16410	7.3915
k23	500	9.1856	2167	5.79E+07	6.302	6.5689
k24	495	8.78E+06	88659	2.01E+09	29711	1063.2
k25	130	6.51E+06	4.19E+06	42.483	8432.4	875.49
k26	110	8.17E+05	9.08E+08	73.402	9.23E+07	1.02E+08
k27	5.00E+06	5.46E+08	7.89E+06	8.39E+09	8.12E+09	6.20E+08
k28	8.00E+05	1.6811	542.13	9.48E+07	6804.8	2.52E+09
k29	500	1.3262	9.18E+05	1.17E+08	4.00E+06	4.34E+08
k30	495	7.02E+06	8497.5	3.29E+05	6.02E+07	27.535

be correlated to how close the fitted parameters are to the actual parameters. The Monte Carlo trials on average are closer to the actual parameters than those predicted by the Genetic Algorithm even though the Genetic Algorithm is much more visually appealing.

Although the parameter fitting was able to produce reasonable approximations of the complex behavior, the underlying kinetic constants that define the network are typically very different than the true kinetic constants. If true kinetic constants need to be obtained, more sophisticated cost fitness measurements, higher data sampling rates, or more advanced algorithms are required. The future of this work involves improving algorithm scalability and capability as well as enhancing computational performance.

CHAPTER IV

COMPARISON OF NUMERICAL INTEGRATION METHODS FOR USE IN SYSTEMS BIOLOGY

4.1 Abstract

Analyses of dynamic network models, which are composed of a system of differential equations, often require the use of numerical integration methods to estimate the networks behavior filling the void left by the lack of an attainable analytical solution. Large systems, as well as moderate-sized, and even small systems of differential equations, often can not be solved analytically. These systems of biochemical differential equations are also especially susceptible to stiffness, a mathematical property that necessitates using a special class of solvers in order to achieve efficient solving. Stiffness arises in systems of differential equations when the integrated variables are changing on very different timescales.

Here, I profile the performance of nine common numerical integration techniques on eight problems obtained from literature. Each of the integrators solved the test problems at different levels of numerical tolerance and was evaluated on two primary criteria, time of execution and time lag error in the obtained solution. I find that no particular integrator outperforms the others under all circumstances but the backward differentiation formulae (BDF) have the best average performance when both stiff and non-stiff problems are considered.

Efficient numerical integration methods are especially important in parameter estimation exercises where a vast number of potential dynamic network models with differing degrees of stiffness are evaluated. The future of systems biology differential equation models greatly depends on harnessing capable numerical methods to solve large problems in a correct and timely manner.

4.2 Background

Although many of the landmark papers on the comparison of numerical integration methods were completed decades ago [23, 30, 38, 86, 37], the rapid advancement of computers, the increasing availability of grid computing resources, and the desire to expand both the size and complexity of dynamic network models have necessitated that commonly employed numerical methods be re-examined. While computing power enhancement is often exclusively viewed in terms of processor speed and Moores Law, the improvement of many other components, such as RAM speed, may cause the rank order performance of specific integration algorithms to change with new generation processors. Due to the varying improvement rate of processing components, algorithms previously not competitive may suddenly outperform yesterday's premier methods. Grid computing is rapidly emerging as a viable scientific resource and necessitating a change in the way algorithms are designed. The advent of grid computing, much like the different evolution times for computing components, has created a potential solution where the yesterday's premier algorithms are no longer competitive with previously inferior algorithms that have been more effectively parallelized for grid usage. Current size of state of the art differential equation models [4] pale in comparison with the size of large linear models of biochemical pathways [74] and the effectiveness of numerical solvers likely plays a role in creating the observed disparity. Dynamic network models are rapidly expanding to include more reactions and components, which in the case of differential equations models may necessitate using a different solving technique because of size limitations or the more likely induction of numerical stiffness.

Numerical integration methods can largely be classified into two distinct categories, stiff and non-stiff solvers. Non-stiff solvers employ explicit methods to solve the initial value problem. The term explicit means that the next point in the solution can be computed by a function that only depends on the previously value of the solution. The function is first evaluated at the initial value and then iteratively proceeds until the end of the integration interval. Explicit integration methodologies mainly vary in their composition of the function to predict the next point in the solution. In this study, we investigate

the following methods: Runge-Kutta with Cash-Karp constants [14], Adams-Bashforth-Moulton [35], and the Bulirsch-Stoer method [90]. Runge-Kutta, including the variant used in this study, combines several algebraic expressions to exactly match a Taylor series expansion of a certain order. The Adams-Bashforth-Moulton is a multistep scheme where first a predictor equation is used to extrapolate the next point in the solution and then a corrector estimates the local truncation error for the iteration. The Bulirsch-Stoer method combines the modified midpoint method and rational extrapolation. This integrator works by first explicitly estimating the point in the solution with several finite step sizes and extrapolating the value of the solution as would be determined by step size of zero [72]. Although integration option not studied here is a Taylor series integrator that calculates the Taylor series of a certain order directly by numerically computing the necessary derivatives at each point in the solution. Recursive programming techniques are employed to defray the high computational cost of the Taylor series integrator [40].

Stiffness can be defined as a property of the set of differential equations where a significant depression in step size is required in some numerical solvers for at least part of the integration interval. Stiffness arises when the solutions of two or more of the differential equations evolve at vastly different timescales [39]. Stiff sets of differential equations can efficiently be solved with implicit methods. Implicit methods differ from comparable explicit methods by requiring the next step in the solution to be a function of not only the previous value of the solution but also a function of the solution value being determined. The stability of each integration step is no longer dependent on the size of the integration step with implicit methods and therefore allows efficient solution of stiff problems [72]. This enhanced stability of implicit methods comes with a cost of decreased solution accuracy as compared to explicit methods. The following implicit solvers were included in this integrator comparison study: backward differentiation formulae (BDF) [26, 35], numerical differentiation formulae (NDF) [85], a Rosenbrock method [84], and semi-implicit extrapolation method [5]. Backward differentiation formulae are implicit predictor-corrector methods that are especially well suited for stiff problems [26]. Numerical differentiation formulae is a recent alteration of the BDF method that allows the truncation error of the solution to

be reduced therefore providing a solution with improved accuracy while still retaining the stability profile of the BDF method [85]. Rosenbrock methods are implicit versions of the Runge-Kutta method. The semi-implicit extrapolation method is a semi-implicit version of the explicit Bulirsch-Stoer method.

Existing software for building and solving models of differential equations in Systems Biology typically employs an integration scheme that is capable of efficiently solving both stiff and non-stiff problems. The most common choice of the integrator for these softwares is LSODA, an integrator that automatically switches between an Adams type method for non-stiff problems and a BDF method for stiff problems [69]. Some of the software packages that use LSODA or similar integration methods include Gepasi [60] and E-Cell [91]. Several software packages also use the similar NDF method such as OBIYagns [43]. Specialized tools, which take advantage of particular mathematical form, have been developed for specific models such as power law systems including PLAS [94].

4.3 Methods

The test problems, integrators, and analysis scripts were all written in the C programming language. The C programming language was selected because of the vast availability of integration methods, its noted performance in numerical methods, and its potential to be used in parallel computing environments including grids. High quality implementations for all of the integration methods were obtained in the C programming language with the exception of numerical differentiation formulae, which was converted from MATLAB 6.5 by the author. The Runge-Kutta with Cash-Karp constants, the Bulirsch-Stoer method, Rosenbrock method, and semi-implicit extrapolation method are all available for purchase at www.nr.com. The Adams predictor-corrector method and backward differentiation formulae are available for free download as part of the SUNDIALS integrator suite at www.llnl.gov/CASC/sundials. The numerical differentiation formulae (NDF) integrator is available as the MATLAB function ode15s in latest version of MATLAB (www.mathworks.com). All of the test problems were also written in the C and MATLAB languages and are in appendix B.

4.4 Results

The first four test problems are examples of classic integrator test problems obtained from previous integrator comparison studies [30]. The fifth test problem models the aerobic oxidation of nicotinamide adenine dinucleotide (NADH) in horseradish [99]. The fifth test problem was included because it is a well studied example of a biochemical system capable of oscillations. The sixth test problem is physiologically based pharmacokinetic model, which details how an intravenous drug disperses and is eliminated from the body [71]. The seventh test problem investigates the signalling and receptor internalization of HER2 receptors [33]. The eighth test problem is recent large-scale metabolism model [4, 3]. Each of the modern numerical integration methods were tested on the entire set of test problems and results are reporting in the following sections.

4.4.1 Test Problem 1: The Oregonator Model

Oregonator is a classical differential equation model that accurately predicts the oscillatory behavior of the Belousov-Zhabotinskii chemical reaction. The organic chemical reaction exhibits limit cycle behaviour characterized by the stable periodic oscillations of intermediate reaction products [24]. The values of the initial conditions, kinetic rate constants, and comparison time as well as the structure of the ordinary differential equations for the three variable system were obtained from [30]. Table 2 shows the results of each of the numerical integrators in terms of execution time and error in the comparison time.

4.4.2 Test Problem 2: High Irradiance Response Model (HIRES)

The HIRES model describes the growth and differentiation of plant tissue under the condition of high levels of irradiance [78]. The model is composed of an eight variable non-linear ordinary differential equation. The structure, initial conditions, equation parameters, and characteristic time value were obtained from [32] and [30]. Table 3 shows the results of each of the numerical integrators in terms of execution time and error in the comparison time.

Table 2: OREGO Results: The Oregonator test problem was simulated from time zero to time 316 seconds. The time column represents the execution time in microseconds. The error column is the percent error of the intergrators result when $y(1)$ reaches the initial condition of 4.0 a second time. The best value for this time is 302.85805 as reported in [30].

Method	Tol = 10^-2	Tol = 10^-2	Tol = 10^-4	Tol = 10^-4	Tol = 10^-6	Tol = 10^-6
	Time (_s)	Error	Time (_s)	Error	Time (_s)	Error
Runge-Kutta	4.3050e+007	0.0103	FAIL	FAIL	FAIL	FAIL
Adams Predictor Corrector	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
Burlisch-Stoer	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
Rosenbrock	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
Backward Differentiation Formulae	1252	17.135	5042	0.0119	10341	0.0028055
Numerical Differentiation Formulae	19722012	-0.3468	726774	-0.31293	152333	-0.31998
Semi-Implicit Exptrapolation	911	-4.82	3604	0.026721	38471	-0.00020362

Table 3: HIRES Results: The HIRES test problem was simulated from time zero to time 400 seconds. The time column represents the execution time in microseconds. The error column is the percent error of the intergrators result when $y(7)$ reaches the same value of $y(8)$, which is 0.00285. The best value for this time is 321.8122 as reported in [30].

Method	Tol = 10^-2	Tol = 10^-2	Tol = 10^-4	Tol = 10^-4	Tol = 10^-6	Tol = 10^-6
	Time (_s)	Error	Time (_s)	Error	Time (_s)	Error
Runge-Kutta	401208	0.010006	399157	-0.0133	412195	0.0068984
Adams Predictor Corrector	125108	0.019328	124117	0.019328	125388	0.019328
Burlisch-Stoer	44734	-0.2062	48876	-0.44989	57901	-0.21152
Rosenbrock	2694	-2.6451	7877	-0.019514	32853	-0.0133
Backward Differentiation Formulae	537	FAIL	1144	-0.29141	2708	0.019328
Numerical Differentiation Formulae	3495	-0.51839	6493	-0.31731	9002	-0.31007
Semi-Implicit Exptrapolation	6383	-8.7805	5681	0.038749	7817	0.020882

Table 4: CLAUS Results: The CLAUS test problem was simulated from time zero to time 40 seconds. The time column represents the execution time in microseconds. The error column is the percent error of the intergrators result when $y(10)$ reaches 90% of its steady state value. The best value for this time is 36.7234 as reported in [30].

Method	Tol = 10^-2	Tol = 10^-2	Tol = 10^-4	Tol = 10^-4	Tol = 10^-6	Tol = 10^-6
	Time (_s)	Error	Time (_s)	Error	Time (_s)	Error
Runge-Kutta	4994	16.481	8206	17.811	16794	17.764
Adams Predictor Corrector	796	17.677	796	17.61	1559	16.702
Burlisch-Stoer	1593	-1.743	1405	17.779	2137	17.616
Rosenbrock	8110	16.28	18769	17.756	52118	17.633
Backward Differentiation Formulae	190	19.359	397	17.53	855	17.705
Numerical Differentiation Formulae	1155	14.43	1885	15.707	3099	15.951
Semi-Implicit Exptrapolation	17099	16.748	14051	17.546	17272	17.683

4.4.3 Test Problem 3: Transient Molecular Flow through a Tube (CLAUS)

The CLAUS problem is partial differential equation model that investigates the transient molecular flow in a tube as treated by Clausing [15]. The partial differential equation model can be discretized into a system of ordinary differential equations as presented by Gottwald and Wanner [30]. The model used for this work was a ten compartment model without chemically reacting flow. The structure, initial conditions, equation parameters, and characteristic time value were obtained from [30]. Table 4 shows the results of each of the numerical integrators in terms of execution time and error in the comparison time.

4.4.4 Test Problem 4: Nerve Excitation Model of Hodgkin and Huxley (HODGK)

The HODGK model is a set of four ordinary differential equations that describes the behavior of the membrane potential and different ionic currents associated with nervous system firing and signal propagation. The first three differential equations describe the potassium activation, sodium activation, and sodium inactivation. The fourth equation describes the membrane potential difference [25]. The structure, initial conditions, equation parameters,

Table 5: HODGK Results: The HODGK test problem was simulated from time zero to time 20 seconds. The time column represents the execution time in microseconds. The error column is the percent error of the intergrators result when $y(4)$ reaches zero. The best value for this time is 8.17888 as reported in [30].

Method	Tol = 10^-2	Tol = 10^-2	Tol = 10^-4	Tol = 10^-4	Tol = 10^-6	Tol = 10^-6
	Time (_s)	Error	Time (_s)	Error	Time (_s)	Error
Runge-Kutta	1594	-8.4902	1289	0.51206	2902	0.068046
Adams Predictor Corrector	1177	-2.8598	1776	-9.3847	1360	-1.3195
Burlisch-Stoer	851	-12.072	841	3.6635	1713	-3.5679
Rosenbrock	598	18.26	5441	7.321	52986	7.321
Backward Differentiation Formulae	238	31.737	584	10.749	1246	3.1325
Numerical Differentiation Formulae	4014	-9.9006	4525	-36.697	2189	-13.462
Semi-Implicit Exptrapolation	242	7.5579	691	7.1595	1546	-0.01757

and characteristic time value were obtained from [30]. Table 5 shows the results of each of the numerical integrators in terms of execution time and error in the comparison time.

4.4.5 Test Problem 5: Aerobic Oxidation of NADH in Horseradish (PO)

Peroxidase-Oxidase reaction involves the oxidation of organic electron donors by molecular oxygen [99]. The catalyst for the reaction is Horseradish Peroxidase (HRP), present in the roots of the horseradish plant, is a cell wall bound enzyme. It consists of a ferric heme group, 308 amino acid residues, two Ca2+, and eight neutral carbohydrate side chains. HRP, in vivo, uses peroxide to oxidize substrates hence this reaction is referred to as peroxidase-oxidase, that is, an oxidation with a peroxidase enzyme as the catalyst. The reaction is modeled as a system of differential equations built from mass action kinetics. This widely studied system has models that widely vary in complexity from several to a vast number of differential equations. Here, we have chose the simplest model that can demonstrate the oscillatory behavior as presented in [48]. The first and second equations represent the rate of concentration changes of oxygen and NADH, respectively. The third and fourth

Table 6: PO Results: The PO test problem was simulated from time zero to time 200 seconds. The time column represents the execution time in microseconds. The error column is the percent error of the intergrators result when $y(1)$, oxygen, reaches 95% of its steady state value. The best value for this time is 82.4159 as determined by high accuracy numerical solution.

Method	Tol = 10^-2	Tol = 10^-2	Tol = 10^-4	Tol = 10^-4	Tol = 10^-6	Tol = 10^-6
	Time (_s)	Error	Time (_s)	Error	Time (_s)	Error
Runge-Kutta	46166	0.001092	56820	-0.017108	58772	-0.017108
Adams Predictor Corrector	FAIL	FAIL	27344	-0.028655	31182	-0.028655
Burlisch-Stoer	FAIL	FAIL	15115	-0.12564	18851	-0.14422
Rosenbrock	147	-0.60514	798	-0.048919	4647	-0.016833
Backward Differentiation Formulae	447	-4.3809	774	-0.028655	1630	-0.011989
Numerical Differentiation Formulae	1099	-1.9398	1892	-1.4112	3482	-1.348
Semi-Implicit Exptrapolation	131	-0.44941	313	-0.045622	1076	-0.20568

differential equations represent the concentration rate of change for a radical form of NAD and radical form of the peroxidase enzyme, respectively. The characteristic time used for this test problem was when oxygen reaches 95% of its steady state value. Table 6 shows the results of each of the numerical integrators when the problem was run from time zero to 100 minutes.

4.4.6 Test Problem 6: Physiologically Based Pharmacokinetics Model (PBPK)

Typical pharmacokinetic models target a specific tissue and investigate the performance of the major pharmacokinetic processes, which are absorption, distribution, metabolism, and elimination. Physiologically-based pharmacokinetic models combine the in vitro classified traditional pharmacokinetic models with in vivo information pertaining to blood volumes and flow rates to create a model that is capable of predicting in vivo behavior of an intravenously injected drug. In this study, we used a thirteen compartment model of diazepam distribution in a 250 gram rat. Each compartment was model with a differential equation that accounted for absorption, elimination, and metabolism. The compartments were arterial blood, venous blood, liver, lungs, adipose, bone, brain, gut, spleen, heart, kidney,

Table 7: PBPK Results: The PBPK test problem was simulated from time zero to time 40 minutes. The time column represents the execution time in microseconds. The error column is the percent error of the integrators result when $y(3)$, liver concentration, reaches a level corresponding to 70% of the peak concentration. The best value for this time is 32.6682 as determined by high accuracy numerical solution.

Method	Tol = 10^-2	Tol = 10^-2	Tol = 10^-4	Tol = 10^-4	Tol = 10^-6	Tol = 10^-6
	Time (_s)	Error	Time (_s)	Error	Time (_s)	Error
Runge-Kutta	38361	-0.020815	39428	-0.0055099	44264	-0.097342
Adams Predictor Corrector	17291	-0.2504	17617	-0.2504	186888	-0.2504
Burlisch-Stoer	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
Rosenbrock	59509	6.1932	408109	0.25468	3932370	-0.082037
Backward Differentiation Formulae	35000	FAIL	29440	-1.0157	35300	-0.2504
Numerical Differentiation Formulae	29357	-2.1882	26662	-1.5552	23564	-1.5552
Semi-Implicit Exptrapolation	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL

skeletal muscle, and skin each represented by differential equations one through thirteen, respectively [71]. The comparison time used for this test problem was the time in which the amount of diazepam 70% of the peak concentration. Table 7 shows the computational time results as well as the error for each of the integrators on this test problem.

4.4.7 Test Problem 7: HER2-mediated Endocytosis Model (HER2)

The family of epidermal growth factor receptors are a set of tyrosine kinases that have been implicated in various cancers most notably breast cancer. In addition to initiating signalling cascades from the plasma membrane, this receptor type is able to leave the cell surface and be transported into the cell via endocytosis. The model presented in [33] investigates the concentration of two of the receptor family subtypes in various forms. These include homodimers, hetereodimers, ligand receptor complexes, antibody bound receptors, and free receptors both inside cells and at the surface of cells. The set of 18 differential equations were integrated from zero to ten minutes. The characteristic time used here was the time at which the concentration of ligand inside the cell reaches 4.75 nM. The timing of each of

Table 8: HER2 Results: The HER2 test problem was simulated from time zero to time 20 minutes. The time column represents the execution time in microseconds. The error column is the percent error of the integrators result when $y(15)$, intracellular ligand concentration, first reaches 4.75 nM. The best value for this time is 7.7620 as determined by high accuracy numerical solution.

Method	Tol = 10^-2	Tol = 10^-2	Tol = 10^-4	Tol = 10^-4	Tol = 10^-6	Tol = 10^-6
	Time (_s)	Error	Time (_s)	Error	Time (_s)	Error
Runge-Kutta	10747	0.025767	12339	0.090183	27607	0.025767
Adams Predictor Corrector	70	FAIL	74	FAIL	970	FAIL
Burlisch-Stoer	1001	-0.25855	1373	-0.48956	2747	-0.14766
Rosenbrock	7586	-0.060122	16549	0.1546	43599	-0.03865
Backward Differentiation Formulae	135	-135.12	135	-135.12	269	-135.12
Numerical Differentiation Formulae	478	49.342	492	-33.342	1202	-22.391
Semi-Implicit Exptrapolation	31270	-0.10307	30279	-0.03865	38052	-0.10307

the integrators as well as the error value for the characteristic time is presented for each of the integrators in Table 8.

4.4.8 Test Problem 8: Yeast Sphingolipid Metabolism Model (SPHINGO)

The SPHINGO model is a set of 25 ordinary differential equations that describe the de novo synthesis of sphingolipids in yeast from precursor products such as fatty acids [4]. The model is built from literature derived reaction information that is converted into generalized mass action (GMA) reaction models [94], which are summed according to network topology to produce rate laws for each of the dependent components. The integration of the system of rate laws allows the concentration of each of the sphingolipids and pathway intermediates to be determined. The structure, initial conditions, and equation parameters were obtained from [4]. The characteristic time of the set of equations was taken to be the point where acetyl CoA reaches 1300mM. This point was determined by solving the system of equations with a very low tolerance. Table 9 shows the results of each of the numerical integrators in terms of execution time and error in the comparison time.

Table 9: SPHINGO Results: The SPHINGO test problem was simulated from time zero to time 50 minutes. The time column represents the execution time in microseconds. The error column is the percent error of the integrators result when $y(25)$, acetyl-CoA concentration, first reaches 1300 mM. The best value for this time is 48.6294 minutes as determined by high accuracy numerical solution.

Method	Tol = 10^-2	Tol = 10^-2	Tol = 10^-4	Tol = 10^-4	Tol = 10^-6	Tol = 10^-6
	Time (_s)	Error	Time (_s)	Error	Time (_s)	Error
Runge-Kutta	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
Adams Predictor Corrector	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
Burlisch-Stoer	FAIL	FAIL	112549684	-0.033828	65654202	0.018771
Rosenbrock	8616	FAIL	57647	FAIL	801717	FAIL
Backward Differentiation Formulae	1980	1.2265	6517	-0.96358	13424	0.10171
Numerical Differentiation Formulae	5751	-2.8185	15160	-2.8185	31070	-2.8185
Semi-Implicit Exptrapolation	4810	FAIL	7424	FAIL	14388	FAIL

4.5 Discussion

The OREGO problem was sufficiently stiff and required many of the integration methods to fail to complete the integration time course. The non-stiff solvers failed in every case except the adaptive step size Runge-Kutta with Cask-Karp constants when a very loose tolerance was employed. The Runge-Kutta method still required an excessive amount of time to complete this integration even at a lose tolerance. The non-stiff solver was two to three orders of magnitude slower than the stiff methods. The Rosenbrock method failed for all tolerance levels. This failure is likely due to the fact that the problem was very stiff instead of a moderately stiff problem, which is typically solved with the Rosenbrock methods. The numerical differentiation formulae (NDF) integrator was the only method to provide an accurate solution at all tolerance levels. The semi-implicit extrapolation and backward differentiation formulae (BDF) integrators provided accurate solutions in a timelier manner than the NDF integrator when the tolerance was below 0.0001. While both are considerably better than the other methods, neither the semi-implicit extrapolation nor

the BDF integrator appear to have a distinct advantage over one another in this test case.

The HIRES test problem was capable of being solved by all of the integration methods regardless of their classification of being stiff or non-stiff solvers. The problem does still appear to have some degree of stiffness as the non-stiff solvers are typically one to two orders of magnitude slower than the comparable stiff methods. The Runge Kutta and Adams solvers both provide solutions accurate to two hundredths of a percent as compared to the actual value at all tolerances. The Burlisch-Stoer solver provides a solution a less accurate solution, half of one percent, in three to tenfold less time as compared to the other non-stiff solvers. The Rosenbrock method delivers a solution that is nearly as accurate as the the Runge Kutta and Adams solvers but with much improved computation speed. The BDF method is again fairly accurate and efficient but fails at the highest tolerance setting. Although the semi-implicit extrapolation integrator appears inferior at low tolerances, the NDF and semi-implicit extrapolation integrators return solutions in a comparable amount of time and with a comparable amount of accuracy for the remaining tolerances. Even with a failure at the crudest tolerance, the BDF method appears to be the preferred method for this problem because of its apparent speed advantage over the other methods while still maintaining accuracy within one percent of the true value.

When the reported time is used for comparison with the integration of the CLAUS problem, none of the integrators appear to perform well with Burlisch-Stoer at crudest error tolerance being the exception. After reinvestigating a figure in the original manuscript the 90% of steady state value time appears to be closer to 30 than the reported 36.7234. All of the integration methods appear to approximate the time as closer to the suspected value shown in the original figure as opposed to the reported value in the text of the original manuscript. The non-stiff methods appear to outperform the stiff methods for each of the comparable types except when comparing the Adams and BDF solvers. The BDF and Adams methods both appear very effective at efficiently and accurately solving the CLAUS problem, when the true time is adjusted. The computational time required for the BDF solver appears to be consistently less than that of the Adams solver.

In analyzing the results of the HODGK test problem, only two of the integration methods, Runge-Kutta and semi-implicit extrapolation, were capable of producing a solution that differed from the reported time value by less than one percent. This accurate solution for each of the methods was only possible at the strictest error tolerance, 0.000001. The Rosenbrock and NDF methods were especially inefficient at solving the HODK test problem both in terms of accuracy and execution speed. The remaining methods were able to execute with sufficient speed but were not able to produce solutions within one percent of the true time at the tolerances tested. The semi-implicit extrapolation solver appeared to provide the best results for the test problem as it was able to produce a solution with more than half of the percent error and with an execution time of half that compared to the second best integrator, the Runge Kutta.

The peroxidase-oxidase (PO) test problem was generally solved with acceptable accuracy by both stiff and non-stiff solvers with the exceptions being the Adams and Burlisch-Stoer methods that fail at the crudest error tolerances. The stiff integrators typically outperform the non-stiff solvers in terms of execution time and have near the same level of accuracy. The NDF method performs the integration with the worst accuracy on average. The Rosenbrock, BDF, and semi-implicit extrapolation integrators all have fast execution times with percent error measurements less than one tenth of one percent at the low and moderate tolerances. BDF, semi-implicit extrapolation, and Rosenbrock solvers are all acceptable choices but semi-implicit extrapolation method performs best over the entire range of tested tolerances.

The physiologically based pharmacokinetic model (PBPK) test problem was not able to be solved by either of extrapolation methods, Burlisch-Stoer and semi-implicit extrapolation, for any error tolerance. The stiff integrators were generally outperformed by their non-stiff counterparts. The Runge Kutta method provided the most accurate solution for all of the tested tolerances. The Adams solver was also quite efficient at providing a reasonable estimate of the solution with an execution time of roughly half of the Runge Kutta at the cruder tolerances. The NDF solver has a faster execution time than the BDF solver but the solution is less accurate in most cases. Although the Rosenbrock method was capable of producing a solution that was among the most accurate when a tighter tolerance was

used, the execution time was noticeably slower than all of the other methods. The Runge Kutta and Adams solvers are the preferred solvers for the PBPK test problem.

In the HER2 test problem, the predictor corrector methods, Adams, BDF, and NDF solvers, were largely ineffective. While the predictor corrector solvers were able to run at execution times not previously seen, the percent error in the comparison time was never lower than 20%, thus providing a largely inaccurate approximation to the true underlying solution. The Rosenbrock and Runge Kutta methods provide accurate solutions in a timely manner with the Rosenbrock method being slightly more efficient at crude tolerances and the Runge Kutta method being superior at tight error tolerances. The extrapolation methods are also capable at providing acceptable solutions but with the Burlisch-Stoer method having slightly more error and the semi-implicit extrapolation solver having a longer execution time. The Runge Kutta solver is the preferred solver for the HER2 test problem because of this consistent performance across all tested tolerances.

The SPHINGO model is the largest of the test problems and provided suitable challenge to all of the solvers. The Runge Kutta, Adams, Rosenbrock, and semi-implicit extrapolation method fail to reach the $y(25)$ level required under all tolerances. The Burlisch-Stoer method provides the most accurate solution but it comes at the cost of very large execution time. The BDF and NDF solvers are quite capable at obtaining a solution in a much more efficient manner as compared to the Burlisch-Stoer solver. The BDF consistently outperforms the NDF method both in terms of accuracy and execution speed for all tested tolerances. While the Burlisch-Stoer method provides an unparalleled accuracy advantage over the other methods, the BDF solver is an acceptable alternative for this problem when decreasing the execution time is a priority.

4.6 Conclusion

No integration method was clearly superior over all of the remaining methods for all of the test problems. The stiff integrators were by in large had faster execution times and less accuracy even when the test problem was termed stiff. The modern Runge Kutta solver, adaptive step size Runge Kutta with Cash Karp constants, was very robust compared

to the other non-stiff solvers and was able to integrate most of the test problems, stiff or not stiff. The efficiency of the Runge Kutta method was often much less than the other methods. The Rosenbrock method, semi-implicit Runge Kutta, appeared to be only marginally more effective and sometimes less effective compared to the non-stiff routine when solving stiff problems. The Burlisch-Stoer method was very capable of providing high accuracy solutions that came at a large execution cost for most of the test problems. The semi-implicit extrapolation solver was decently robust and very effective in solving certain test problems. The predictor corrector solvers were on average the best performing class of solvers. Even though the NDF method was created to be more efficient than the older BDF method, the high quality implementation of the BDF method in CVODE not only made up the lost computational efficiency but also provided a more accurate solution for most of the test problems. The Adams method was quite capable providing high quality solutions for non-stiff problems.

Overall, the CVODE implementation of the Adams and BDF solvers is the logical first choice for an integrator. This implementation provided a timely solution with sufficient accuracy in most of the test problems as well as having built in parallel computing potential. For parameter estimation in systems of differential equations that model biochemical processes, there is no substitute for preparing a Monte Carlo test case for the integration method using the expected ranges of the parameters. In certain cases, multiple integration methods might have to be used to get an accurate approximation of the true behavior of the biologically inspired system of differential equations.

CHAPTER V

A HIGH PERFORMANCE COMPUTING SOLUTION TO PARAMETER ESTIMATION IN METABOLIC NETWORKS

5.1 Abstract

The primary goal of this work is to develop a high performance-computing framework to extend previous metabolic pathway parameter estimation techniques for use in larger, more complex pathway models. The high performance computing solution is also beneficial to smaller networks as the parameter estimation can be completed in a timelier manner thus allowing more time to do replicate fittings to boost parameter confidence. The current paradigm for achieving this goal is enhancing several previously implemented optimization routines with the standard message passing interface library (MPI) allowing the routine to run on a distributed set of multiprocessors, a commodity cluster. In this current work, we concentrate on improving the speed parameter estimation in smaller networks with the hope of perfecting these techniques before moving to larger systems. We construct a simulated six node network with 30 parameters. The parameter estimation is performed with an MPI-enhanced Monte Carlo technique, and an MPI-enhanced Genetic Algorithm.

This work considers hypothetical six node five connection network where the underlying kinetic parameters that determine the action in the network are known. The two aforementioned optimization routines are applied to this hypothetical network in order to elucidate the known parameters. The predictability of network action and of the accuracy of kinetic parameters themselves is discussed along with computational effort necessary to generate the results. The current method did not provide suitable parameters for building a large-scale predictive solution. The hope is that this small-scale approach on synthetic data can be enhanced and combined with other computational methods to accurately estimate

the necessary parameters to build an accurate large-scale simulation of multiple component cellular process.

5.2 Background

In modern life science research as well as many other areas, the computational problems are becoming increasing larger and more complex at astounding rate. While Moores law has shown that computing speed doubles every eighteen months, common scientific computational tasks in today's world to be completed in a timely manner still require more processing power than the average central processing unit (CPU) can provide. In previous generations, this discrepancy in computing power was often addressed by the use of super computers [73]. Super computers, expensive specialized hardware developed oftentimes for a specific scientific computing task, have been employed to solve difficult problems since the days of World War II [73]. The super computing model has been enhanced and evolved greatly over the past 50 years but until recently cost concerns have prevented the average scientific researcher from harnessing super computing power to solve their computational problems. Only in recent years, programming standards and hardware manufactures have opened a new avenue to achieve super computing performance at a fraction of the cost [73]. This new computing model involves effectively dividing computational tasks into smaller components that can be run in parallel to each other on a set of commodity computers connected with an interconnect. Each of these subtasks can communicate and trade data with the other tasks to create a superior computation vehicle [73].

The message passing interface (MPI) is a standard communication library for C and FORTRAN programming languages. This library allows serial codes developed in C or FORTRAN to run simultaneously across distributed systems. The primary reason for adhering to the MPI Standard, other than ease of development, is that the source code becomes portable as it can be run on a diverse set of hardware architectures. Thus, allowing source developed on today's machines to run on tomorrow's faster parallel computers.

5.3 Methods

The overarching goal of this work and the work that preceded it is to develop a method to determine kinetic parameters of enzymatic biochemical reactions for use in large scale metabolism simulation networks. The current methodology for achieving this larger goal involves completing a combination of experimental and computational steps. This work focuses on the investigating the computational objective, determining kinetic parameters from time course experimental data. Here, we use various two global optimization routines to provide an estimate of the kinetic parameters by producing a best fit of the simulated network. This section highlights the differences in the basic system design imperative to the high performance computing implementation.

5.3.1 Integration Method

The integrator module in the workflow is the odes15s integrator from the Matlab ODE Suite [85]. This integrator is valid for both stiff and non-stiff sets of ordinary differiential equation initial value problems. The integrator uses a combination of Backward Differentiation Formulas (BDFs), as known as Gears Methods, and Numerical Differentiation Formulas (NDFs) [85]. Preliminary trials showed that vast range of kinetic parameters required that a stiff integrator be used. Even with a satisfactory stiff integrator, points of excessive stiffness still were attainable using global search methods so the integrator was altered to eliminate points on the optimization surface that required excessive computational effort. The vital components of Matlabs ode15s for integrating the type of differential equation specific to enzymatic reactions were converted to C to allow efficient incorporation into a C/MPI code.

5.3.2 Monte Carlo

The Monte Carlo methods employed in this work consist of picking a vector of random numbers that represent the kinetic rate constants. A uniform distribution from zero to one is returned for each element of the vector. This number between zero and one directly maps to a rate constant between 0.1 and 109. Monte Carlo methods have been applied in many

problems, most notably integration.

Monte Carlo methods offer a random sample of the entire search space. The principle behind this tool is that a sufficiently large random sample will be to converge to accurately represent the entire search space. In addition to gathering population statistics in a much timelier manner than exhaustive brute force searching, sufficient Monte Carlo sampling allows a reasonable approximation for the global extremes but does not guarantee that the true global minimum will be found.

Because of the obvious independence of individual random samples from each other, Monte Carlo methods are trivially and effectively converted into parallel computing methods. Extra caution only needs to be taken to insure that the same sequence of random numbers is not used on each processor.

5.3.3 Genetic Algorithm

The genetic algorithm, initially conceived by John Holland, is population-based routine rooted in the evolution theory of Charles Darwin [36]. The genetic algorithm mimics the principles of natural selection through a series of operators defined in computer code. Selection, crossover, mutation, and inversion were the basic operators employed in Holland's original formulation. Most current genetic algorithm implementations, including the one discussed in this work, only employ the selection, crossover, and mutation operators [63]. For more on Genetic Algorithm basics see section 1.4.6.

Parallelizing the genetic algorithm is much more conceptually challenging than Monte Carlo methods. An extensive amount of research has gone into developing competent schemes where genetic algorithms achieve speedup [10]. Since genetic algorithms are random processes, parallelization should increase the chance of discovery in the same way that replicate runs do but mathematical proof of insured speedup still has yet to be found [10]. The genetic algorithm employed here is equivalent of running replicate genetic algorithms with different random states.

5.4 Results and Discussion

This work was comprised of three sections with each section further extending the work completed in the previous sections. First, the integrator performance was investigated. Secondly, the results of the parallel Monte Carlo trials are discussed. Finally, a parallel genetic algorithm implementations performance is presented.

5.4.1 Integrator Performance

A study in integrator reliability and performance is warranted because the integrator used in this work required language translation to enable its use in a multiprocessor environment. The converted source underwent several quality control metrics. The first one insured that a comparable number of steps are taken by both the original Matlab implementation and the C version. Figure 23 shows the value of the step size for the entire integration of a sample problem for the C and Matlab integrators.

Figure 23 shows that the steps each integrator implementation takes to complete the integration is essentially the same given the difference in working precision. The sample problem is a rather simple integration so slightly more variation is likely to exist in more difficult stiff problems but this increased difference is likely still insignificant. In addition to the step size, the actual integrated values in time were also investigated. Figure 24 is representative example of how the exact values of one of the variable in the system of equations in the integration compare between the implementations.

Figure 24 demonstrates the C implementations ability to reproduce the value of a variable in time compared with the Matlab implementation. While the minimal variation can be seen in the above figure, the following two figures accentuate the differences between the two implementations. Figure 25 displays the absolute difference while Figure 26 displays the percent error assuming the Matlab implementation value is the actual.

Figures 23 – 26 show conclusion proof that the C implementation is able to mimic the behavior of the Matlab standard function. This proves the integrator translation is capable at replacing the Matlab function used in the previous workflow. Many other sample problems of varying degrees of difficulty were also checked for accuracy in the same manner

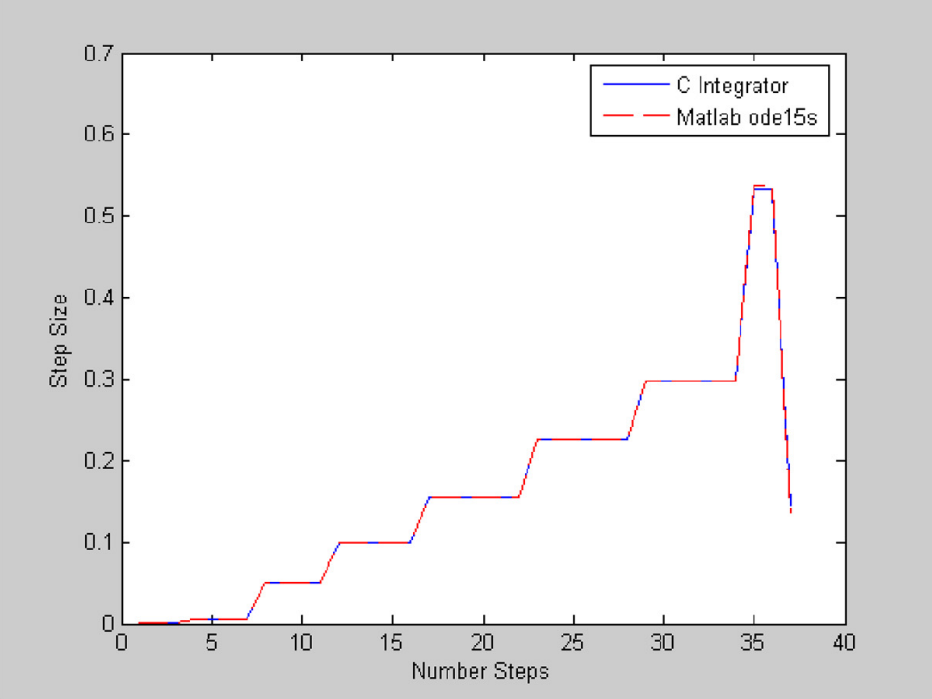

Figure 23: Integrator Step Size Comparison: The step size of each integrator implementation is shown in the figure. The C integrator 37 steps while the original Matlab ode15s solved the problem in 38 steps.

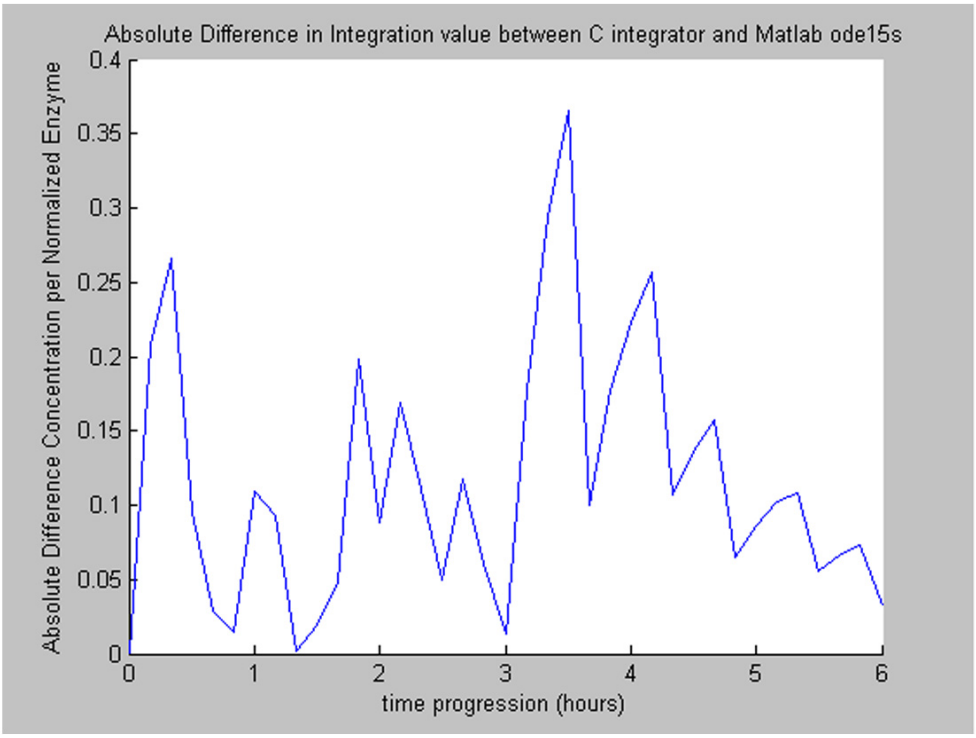

Figure 24: Integration Value Comparison: The values in time that are predicted by each integrator implementation for a representative variable in the system of equations integrated.

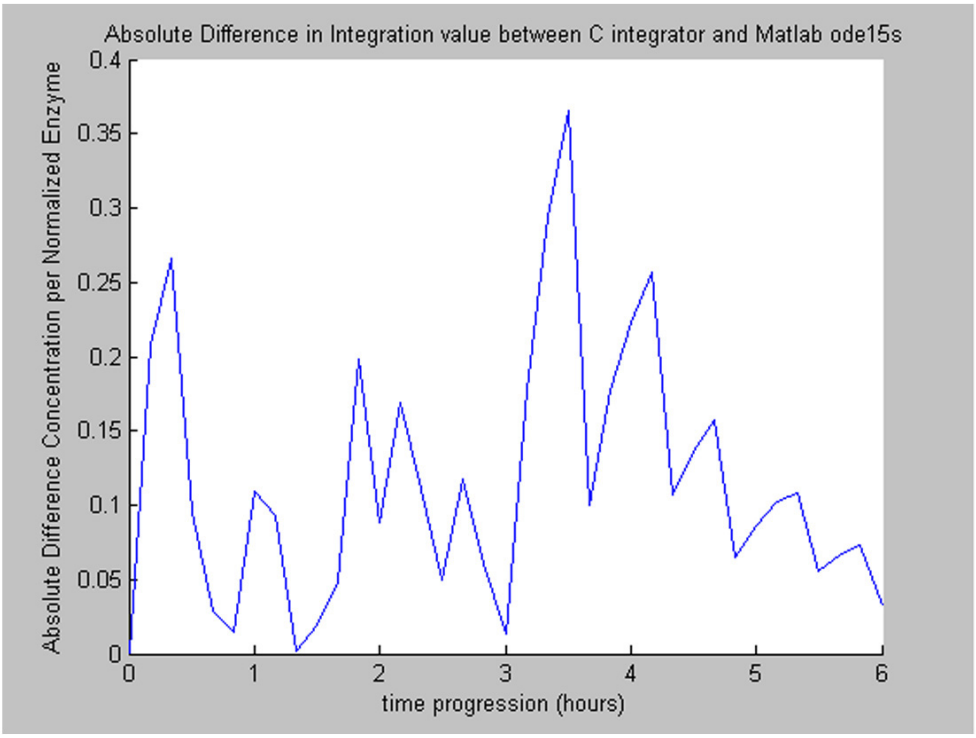

Figure 25: Integrator Absolute Difference: The absolute difference for a representative integrated variable from the sample system of equations ranges from zero to 0.36.

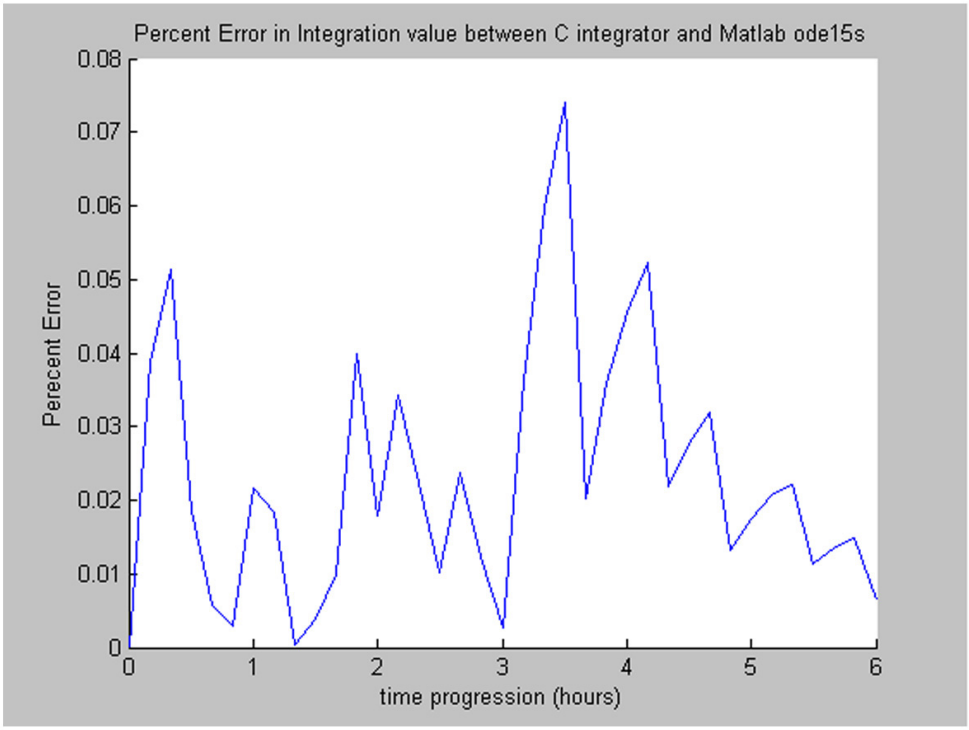

Figure 26: Integrator Percent Error: The percent error for a representative integrated variable from the sample system of equations ranges from zero to roughly 0.08%. This implies for this particular value the C integrator is at least 99.92% accurate in producing the same value as the Matlab function.

as the previous sample problem. The results of these screenings are similar suggesting and appear to be without systematic error that a robust translation has been made.

Now that the implementation has proven competent the performance must be gauged to insure computational time is not being wasted. Although the Matlab programming language is quite efficient for certain types of problems, the source scripts are interpreted on the fly as the program executes. Typically, interpreted languages lag behind compiled languages so performance improvement would not be shocking. The Matlab code is of commercial quality so many source optimizations have been likely been done to enhance performance. The goal of the source conversion is to have the code execute with at least the same speed of the interpreted source. Figure 27 shows the performance of three integrator implementations: the original ode15s, a stripped down Matlab version of ode15s, and the C implementation of the stripped down source.

As Figure 27 depicts, the C integrator was capable of considerable speedup versus either of the Matlab implementations. All of the lines are linear because an identical problem was solved 1,000 times. Although these measurements were made on different operating systems, the results are indicative of what could be expected on any platform. The processor clock speed of the machine used to get the C benchmark was only roughly 60% of the clock speed of the machine for the Matlab benchmarks.

In order to insure the observed speedup was not just an artifact of solving a particular problem, a set of 100 randomly chosen integration problems was prepared. Figure 28 shows the computing time required to solve the 100 random integrations for all three of the implementations.

While speedup of the integration with the C integrator is still apparent, the amount of difference between the integrators is not as large as previous tests indicate. Because the 100 sample problems are chosen at random, the results of this test may vary with every trial but the C integrator on average appears to be roughly five times faster.

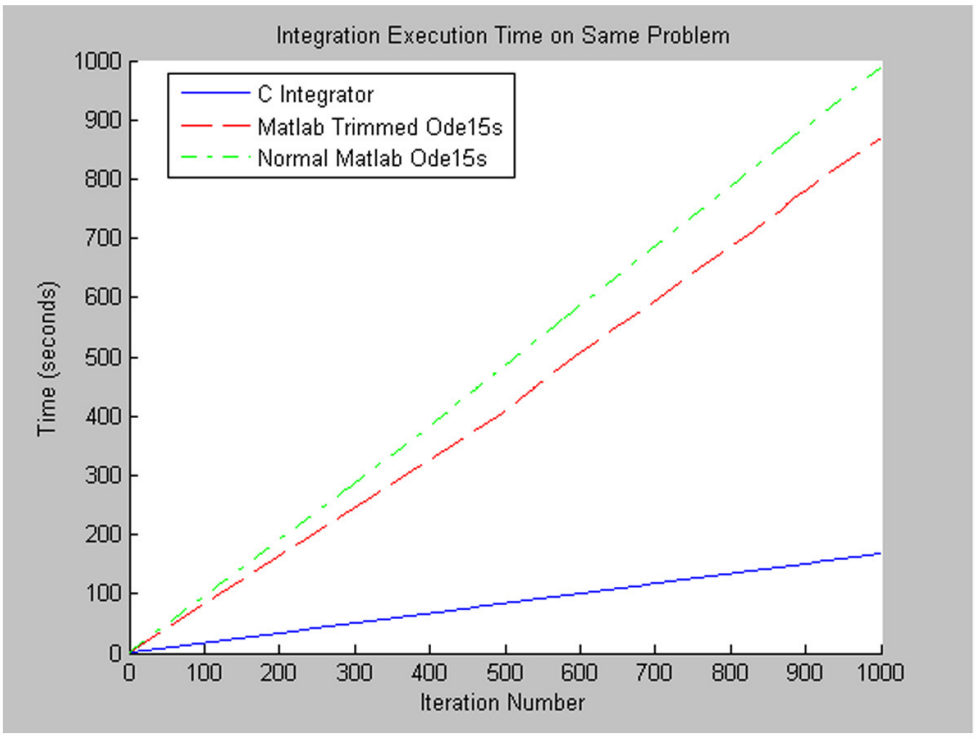

Figure 27: Integrator Performance on Non-stiff problem: The three integrator implementations execution times were gathered from solving the same non-stiff system of ordinary differential equations 1,000 times.

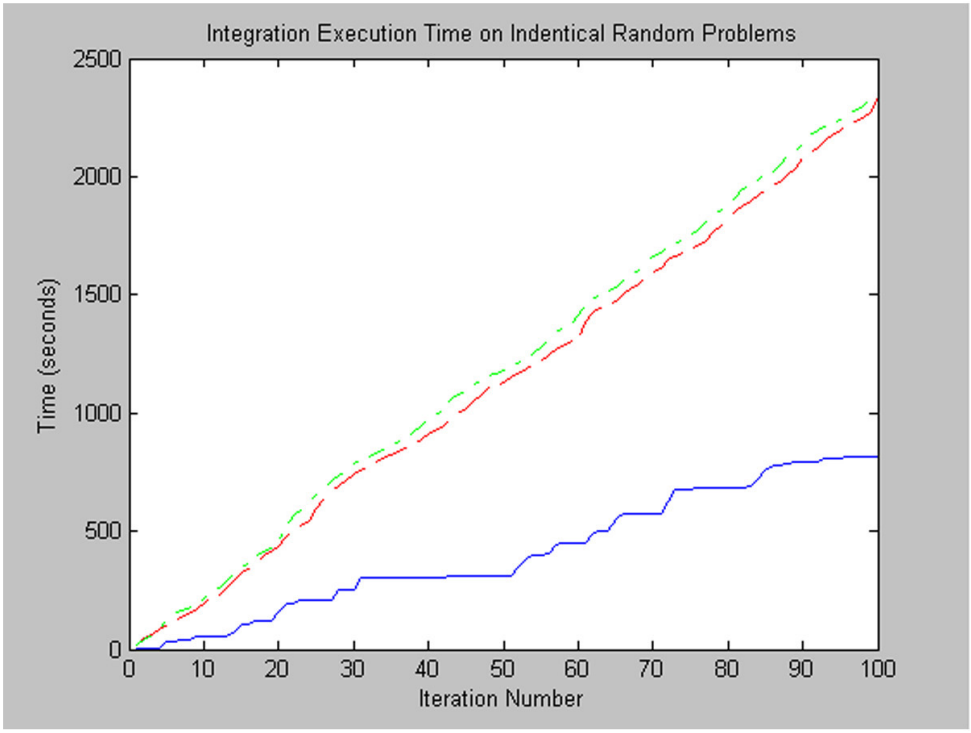

Figure 28: Integrator Execution Time 100 Random Problems: Each of the integrator implementation completed the same 100 problems that were randomly generated. The C integrator also allowed the integration of problems that produced negative mass to be terminated as soon as the mass of one chemical species became negative.

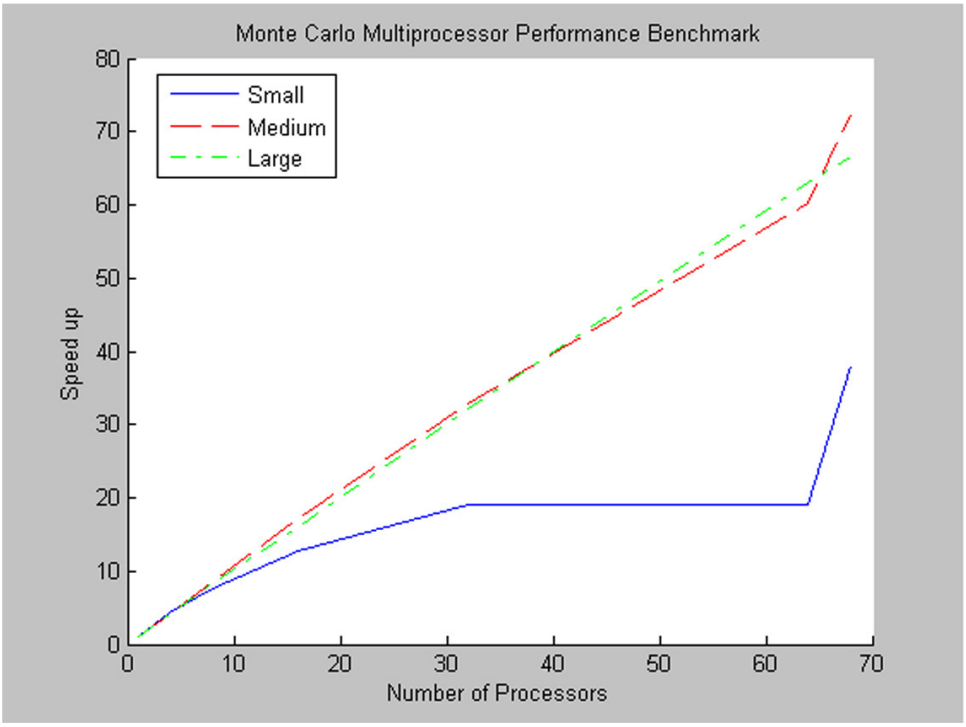

Figure 29: Parallel Monte Carlo Speed Up as a function of Number of Processors: The data for the figure was obtained on a cluster of 34 HP rx-2600 dual processor nodes. The small problem contained 1,000 trials and executed in roughly seven seconds on a single processor. The medium size was ten times larger than the small problem with 10,000 iterations. The large problem was ten times larger than the medium problem with a total iteration count of 100,000. Each of the data points, gathered at 1, 4, 8, 16, 32, 64, and 68 processors, is an average execution time over six trials.

5.4.2 Monte Carlo Trials

Several Monte Carlo trials were performed to gauge the potential speed up due to parallelization and also assess the parallel Monte Carlos ability to predict parameters in the given test problem.

Figure 29 displays speedup, the serial execution time over the parallel execution time. While the two smaller problems deviate from the linear trend between 64 and 68 processors, linear speed is observed. Although less likely a measurement artifact because of replicates, the sudden jump in speedup is likely due to the precision of the timing device. Perfect speedup of a Monte Carlo process would give a straight line with a slope of one. In this

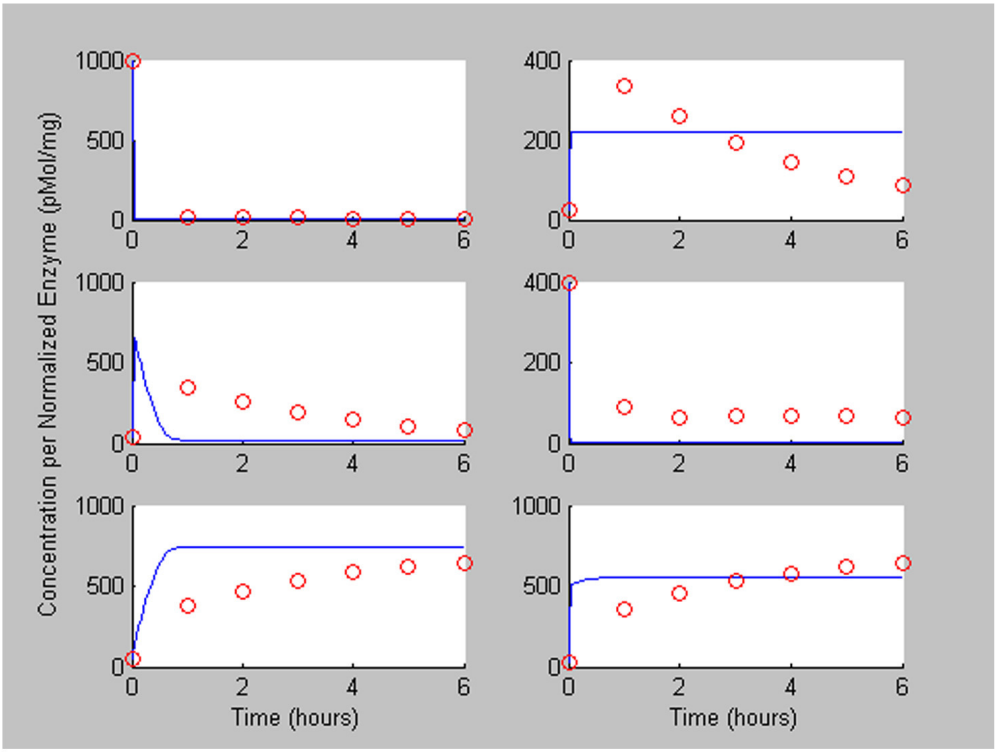

Figure 30: Parallel Monte Carlo best fit: The Monte Carlo scheme completed in 4.5 hours

work, the slope of the line for the largest tested sample problem is roughly .97, which implies a speed of approximately 66 times on 68 nodes. This study shows the same type of parallel efficiency that previous work involving parallel Monte Carlo codes has shown.

Although the parallel Monte Carlo codes proved computationally efficient, the parallel methods ability to find acceptable parameters to fit the data effectively still needs to be investigated. Figure 30 below shows the best fit of a single parallel Monte Carlo trial with 34 million iterations.

5.4.3 Genetic Algorithm Trials

The performance of the genetic algorithm was also evaluated in the same that the Monte Carlo method was but due to the heterogeneity of replicate runs speed of graphs were not obtained. Genetic Algorithms are largely probabilistic serial events so a measure of the

amount of quality improvement due parallelization is difficult to define. When the parallel code was executed for 10,000 generations with a population size of 50, single processor completion times varied between 100 and 1,000 times different. Because of the large difference in computing time between processors, a simplistic load balance scheme was devised to more effectively use the computational power. This scheme affords the processors that finish quickly the ability to reinitialize and complete a second trial while the program is still waiting for the other divisions to finish. Once the slowest division finishes a termination signal is sent the processors that have been reinitialized. Typically, faster processors can complete their work and be reinitialized up to 20 times before the algorithm terminates. This scheme still does not completely rid the algorithm of efficiency problems but it is considerably more effective than the initial parallelization.

The load balanced parallel genetic algorithm was used to provide parameter estimates for the same sample problem used in the Monte Carlo trial. The best-fit results of this trial are show in Figure 31.

Although the parallel genetic algorithm process produced mediocre fitting results, the method establishes an entry point for more complicated methodologies that can be pursued in the refinement of the current work. The genetic algorithm accomplished a best-fit that is similar to one found using 46 hours of parallel Monte Carlo. More research most go into choosing mutation rates and other genetic algorithm parameters to ensure a high quality parameter estimates are made for all types of problems.

5.5 Conclusion

While the scientific results of these parallel endeavors are minimal, significant research progress was made in equipping the parameter estimation method with more computing power. Even though the immediate benefits of the work are not apparent, the vastly augmented computational power and portability this work affords the previously developed methodology will aid in the methods ability to solve more complex problems as scientific refinements to the method are completed.

There are numerous avenues of future work. Computationally, more can be done to load

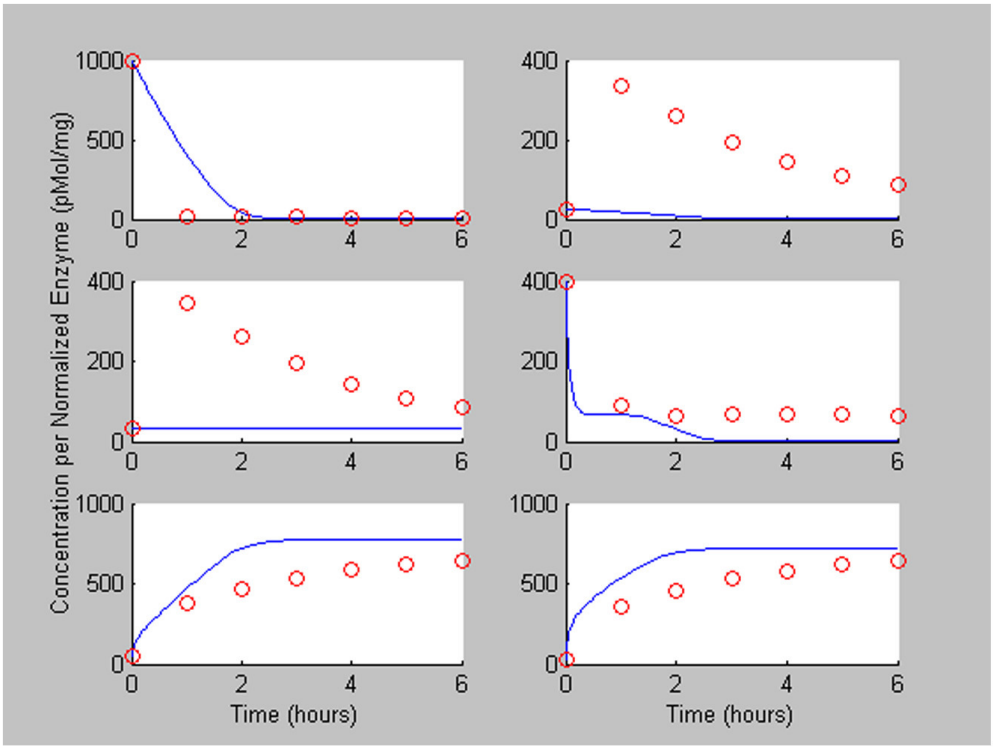

Figure 31: Parallel Genetic Algorithm Best Fit: The parallel genetic algorithm trial used to create this fit ensured that each processor completed at least one 10,000-generation run with a population size of 50. The process took about 27 hours on 34 CPUs.

balance the genetic algorithm to ensure better speedup. Local optimization routines used in conjunction with either the Monte Carlo search or the genetic algorithm can be explored. Scientifically, more research must be put into the studying the behavior of the set of differential equations under various conditions and into developing methods to intelligently reduce the parameter space.

APPENDIX A

MATLAB CODES USED IN MODELING FRAMEWORK

This chapter contains the Matlab implementations of the components of the Modeling Framework.

A.1 Cost Function

```
function cost=kinetics_costFunNew (k, X, F, R, Et)
%KINETICS_COSTFUN Sum of Absolute Residuals Cost Function
%   COST=KINETICS_COSTFUN(k,X,F,R) returns a scalar cost that is a metric
%   of the how well a set of reaction constants fits the experimental
%   dataset. X is a matrix containing experimental data includign time
%   points and measurements (size time points x nodes). k is a vector of
%   the kinetic parameters sent fromt he optimization functioon. F is the
%   nxm footprint matrix which describes how the networks n nodes are
%   connected via m reactions. R is the lxm reaction description matrix
%   which gives the type of reaction for each of the columns in F.
for i = 1:length(k),
    if (k(i) < 0) || (k(i) > 1)
        cost = 15;
        return
    end
end
sizeX = size(X);
time= X(:,1);
Y_zero = X(1,2:sizeX(2))';
OPTIONS = [];
```

```
for i=1:length(time),
    timespan(i)=X(i,1);
end
[t,Y] = ode15Ps('kinetics_odefunNew',timespan,Y_zero,OPTIONS,k,F,R, Et);
sizeY = size(Y);
if t(length(t)) ~= timespan(length(timespan))
    Y_save(1,1) = -909.00;
else
    Y_save = Y;
end
if Y_save(1,1) == -909,00;
    cost = 10;
    return
%elseif sum(Y_save < 0) > 0
    %cost = 10^10;
    %return
else
    for i=1:length(Y_zero),
        cost_temp(i) = 0;
        for j=2:length(time),
            %if (X(j,i+1) ~= 0)
                cost_temp(i) = cost_temp(i) + (X(j,i+1)-Y_save(j,i))^2;
            %else
                %cost_temp(i) = cost_temp(i)+ ((abs(X(j,i+1)-Y_save(j,i)))/1) * 100;
            %end
        end
    end
end
cost = log10(sum(cost_temp));
```

A.2 ODE Constructor

```
function deriv = kinetics_odefunU(t,y,OPTIONS,k,F,R,Et)
%KINETIC_ODEFUN General Function representing a system of enzymatic
%reactions
%   DERIV = KINETIC_ODEFUN(t,y,k,F,R) returns a nxl vector that represents
%   the derivatives of a system of kinetic equations. t is a scalar time
%   point at which the derivative system is evaluated. y is an nxl vector
%   representing the current concentration of each of the n species. k
%   represents the vector of kinetic rate constants being fit by the
%   optimization function size varies based on R. F is the nxm footprint
%   matrix which describes how the networks n nodes are connected via m
%   reactions. R is the lxm reaction description matrix which gives the
%   type of reaction for each of the columns in F.
neq = length(y);
ncn = length(R);
kstart = 1;
for i=1:ncn,
    A=0;
    Aset = 0;
    B=0;
    P=0;
    Pset = 0;
    Q=0;
    numSvar = 0;
    numPvar = 0;
    for j=1:neq,
        if(F(j,i)== -1) & (Aset == 0)
            A=y(j);
            Aset = 1;
```

```
        numSvar = 1;
    elseif(F(j,i)== -1) & (Aset ~= 0)
        B=y(j);
        numSvar = 2;
    end
    if(F(j,i)== 1) & (Pset == 0)
        P=y(j);
        Pset = 1;
        numPvar = 1;
    elseif(F(j,i)==-1) & (P~=0)
        Q=y(j);
        numPvar = 2;
    end
end
decide = R(i);
if(decide == 0)
    %Constant Flux (Zero Order)
    k_fun = k(kstart);
    reactionVelocity(i) = Reaction_ZeroOrderU(k_fun, Et(i));
    kstart = kstart+1;
end
if(decide == 1)
    %Mass Action Flux
    k_fun = k(kstart);
    reactionVelocity(i) = Reaction_MassActionU(k_fun, A, B, numSvar);
    kstart = kstart + 1;
end
if(decide == 2)
    %Reversible Mass Action
```

```
    k_fun = k(kstart:(kstart+1));
    reactionVelocity(i) = Reaction_RevMassActionU(k_fun, A, B,
    P, Q, numSvar, numPvar);
    kstart = kstart + 2;
end
if(decide == 3)
    %Michaelis Menten Single Substrate Flux
    k_fun = k(kstart:(kstart+2));
    reactionVelocity(i) = Reaction_MichaelisMU(k_fun , A, Et(i));
    kstart = kstart + 3;
end
if(decide == 4)
    %Single Substrate Reversible Michaelis Menten Flux
    k_fun = k(kstart:(kstart+3));
    reactionVelocity(i) = Reaction_RevMichaelisMU(k_fun, A, P, Et(i));
    kstart = kstart + 4;
end
if(decide == 5)
    %Uni Uni Flux Rate
    k_fun = k(kstart:(kstart+5));
    reactionVelocity(i) = Reaction_UniUniU(k_fun, A, P, Et(i));
    kstart=kstart + 6;
end
if(decide == 6)
    %GMA Flux
    k_fun = k(kstart:(kstart + numSvar));
    reactionVelocity(i) = Reaction_GMAU(k_fun, A, B, numSvar);
    kstart = kstart + (numSvar + 1);
end
```

```
    if(decide == 7)
        %Reversible GMA Flux
        k_fun = k(kstart:(kstart + numSvar + numPvar + 1));
        reactionVelocity(i) = Reaction_RevGMAU(k_fun, A, B, P, Q, numSvar, numPvar);
        kstart = kstart + (numSvar +  numPvar + 2);
    end
end
deriv_temp=sum((F*diag(reactionVelocity))');
deriv=deriv_temp';
```

A.3 Zero Order Reaction

```
function velocity = Reaction_ZeroOrderU(kfun, Et)
%Zero Order Reaction
velocity = kfun;
end
```

A.4 Mass Action Reaction

```
function velocity = Reaction_MassActionU(kfun, A, B, numSvar)
%Mass Action Reaction
%    if (kfun(1) <= .25)
%        k_scaled(1) = kfun(1) / .25 * 1e-2;
%    elseif (kfun(1) <= .50)
%        k_scaled(1) = (kfun(1) - .25) / .25 * 1e-1;
%    elseif (kfun(1) <= .75)
%        k_scaled(1) = (kfun(1) - .50) / .25;
%    else
%        k_scaled(1) = (kfun(1) - .75) / .25 * 1e1;
%    end
    k_scaled(1) = kfun(1);
```

```
% if (numSvar == 1)
velocity = k_scaled(1) * A;
% elseif (numSvar == 2)
% velocity = k_scaled * A * B;
%    end
end
```

A.5 Reversible Mass Action Reaction

```
function velocity = Reaction_RevMassActionU(kfun, A, B, P, Q, numSvar, numPvar)
%Reversible Mass Action Reaction
%    for i=1:2,
%        if (kfun(i) <= .25)
%            k_scaled(i) = kfun(i) / .25 * 1e-2;
%        elseif (kfun(i) <= .50)
%            k_scaled(i) = (kfun(i) - .25) / .25 * 1e-1;
%        elseif (kfun(i) <= .75)
%            k_scaled(i) = (kfun(i) - .50) / .25;
%        else
%            k_scaled(i) = (kfun(i) - .75) / .25 * 1e1;
%        end
%    end
k_scaled(1) = kfun(1);
k_scaled(2) = kfun(2);
% if (numSvar == 1)
velocityF = k_scaled(1) * A;
% elseif (numSvar == 2)
% velocityF = k_scaled(1) * A * B;
%    end
% if (numPvar == 1),
```

```
velocityR = k_scaled(2) * P;
% elseif (numPvar == 2)
% velocityR = k_scaled(2) * P * Q;
%    end
    velocity = velocityF - velocityR;
end
```

A.6 Michaelis Menten Reaction

```
function velocity = Reaction_MichaelisMU(k , S, Et)
%Three Parameter Michaelis Menten Model
%    if(k(1) < .33333)
%        k1 = k(1)/.33333 * 10^6;
%    elseif(k(1) < .66666)
%        k1 = (k(1)-.33333)/.33333 * 10^7;
%    elseif(k(1) < .99999)
%        k1 = (k(1) - .66666)/.33333 * 10^8;
%    else
%        k1 = 10^8;
%    end
%    k1 = k1 * 3600 * 10^-12;
%    if(k(2) < .25)
%        kn1 = k(2)/.25 * 10^1;
%    elseif(k(2) < .50)
%        kn1 = (k(2)-.25)/.25 * 10^2;
%    elseif(k(2) <= .75)
%        kn1 = (k(2) - .50)/.25 * 10^3;
%    elseif(k(2) > .75)
%        kn1 = (k(2) - .75)/.25 * 10^4;
%    end
```

```
%    kn1 = kn1 * 3600;
%    if(k(3) < .20)
%        k2 = k(3)/.20 * 10^1;
%    elseif(k(3) < .40)
%        k2 = (k(3)-.20)/.20 * 10^2;
%    elseif(k(3) < .60)
%        k2 = (k(3) - .40)/.20 * 10^3;
%    elseif(k(3) <= .80)
%        k2 = (k(3) - .60)/.20 * 10^4;
%    elseif(k(3) > .80)
%        k2 = (k(3) - .80)/.20 * 10^5;
%    end
%   k2 = k2 * 3600;
    k1 = k(1);
    kn1 = k(2);
    k2 = k(3);
    KM = (kn1 + k2) / k1;
    Vmax = k2 * Et;
    velocity = (Vmax * S) / (S + KM);
end
```

A.7 Reversible Michaelis Menten Reaction

```
function velocity = Reaction_RevMichaelisMU(k , S, P, Et)
%Four Parameter Reversible Michaelis Menten Model
%    if(k(1) < .33333)
%        k1 = k(1)/.33333 * 10^6;
%    elseif(k(1) < .66666)
%        k1 = (k(1)-.33333)/.33333 * 10^7;
%    elseif(k(1) < .99999)
```

```
%        k1 = (k(1) - .66666)/.33333 * 10^8;
%    else
%        k1 = 10^8;
%    end
%    k1 = k1 * 3600 * 10^-12;
%    if(k(2) < .25)
%        kn1 = k(2)/.25 * 10^1;
%    elseif(k(2) < .50)
%        kn1 = (k(2)-.25)/.25 * 10^2;
%    elseif(k(2) <= .75)
%        kn1 = (k(2) - .50)/.25 * 10^3;
%    elseif(k(2) > .75)
%        kn1 = (k(2) - .75)/.25 * 10^4;
%    end
%    kn1 = kn1 * 3600;
%    if(k(3) < .20)
%        k2 = k(3)/.20 * 10^1;
%    elseif(k(3) < .40)
%        k2 = (k(3)-.20)/.20 * 10^2;
%    elseif(k(3) < .60)
%        k2 = (k(3) - .40)/.20 * 10^3;
%    elseif(k(3) <= .80)
%        k2 = (k(3) - .60)/.20 * 10^4;
%    elseif(k(3) > .80)
%        k2 = (k(3) - .80)/.20 * 10^5;
%    end
%    k2 = k2 * 3600;
%    if(k(4) < .20)
%        kn2 = k(4)/.20 * 10^1;
```

```
%    elseif(k(4) < .40)
%        kn2 = (k(4)-.20)/.20 * 10^2;
%    elseif(k(4) < .60)
%        kn2 = (k(4) - .40)/.20 * 10^3;
%    elseif(k(4) <= .80)
%        kn2 = (k(4) - .60)/.20 * 10^4;
%    elseif(k(4) > .80)
%        kn2 = (k(4) - .80)/.20 * 10^5;
%    end
%    kn2 = kn2 * 3600 * 10^-12;
    k1 = k(1);
    kn1 = k(2);
    k2 = k(3);
    kn2 = k(4);
KMS = (kn1 + k2) / k1;
KS = (k1 * k2) / (kn1 + k2);
KMP = (kn1 + k2) / kn2;
KP = (kn1 * kn2) / (kn1 + k2);
velocity = ((KS * Et * S) - (KP * Et * P))/(1.0 + S/KMS + P/KMP);
end
```

A.8 Uni Uni Reaction

```
function velocity = Reaction_UniUniU(k,S,P,Et)
% KINETIC_UNIUNI Enzymatic Reaction Model
%   VELOCITY=KINETIC_UNIUNI(K,S,P) gives the accumulation of the product
%   species as a function reaction rate constants (k -- 6x1 vector),
%   substrate concentration (S -- 1x1 value ) and product concentration ( P
%   -- 1x1 value).
%   References : [1] EL King and C Altman, A schematic Method deriving
```

```
%   the rate laws for enzyme catalyzed reactions,
%   Journal of Physical Chemistry, 60 (1956), pp1375-1378
%   [2] V Leskovac, Comprehensive enzyme kinetics,
%   Kluwer Academic/Plenum Pub, New York, 2003
%   [3] K.M. Plowman, Enzyme Kinetics, McGraw-Hill, New York,1971
    %for i=1:6,
        %if(k(i) < .11)
            %k_unscaled(i)=k(i)*10^1;
        %elseif(k(i)<.21)
            %k_unscaled(i)=(k(i)-.1)*10^2;
        %elseif(k(i)<.31)
            %k_unscaled(i)=(k(i)-.2)*10^3;
        %elseif(k(i)<.41)
            %k_unscaled(i)=(k(i)-.3)*10^4;
        %elseif(k(i)<.51)
            %k_unscaled(i)=(k(i)-.4)*10^5;
        %elseif(k(i)<.61)
            %k_unscaled(i)=(k(i)-.5)*10^6;
        %elseif(k(i)<.71)
            %k_unscaled(i)=(k(i)-.6)*10^7;
        %elseif(k(i)<.81)
            %k_unscaled(i)=(k(i)-.7)*10^8;
        %elseif(k(i)<.91)
            %k_unscaled(i)=(k(i)-.8)*10^9;
        %elseif(k(i)<= 1)
            %k_unscaled(i)=(k(i)-.9)*10^10;
        %end
    %end
    %k_unscaled(1) = k_unscaled(1) * 3600 * 10^-12;
```

```
    %k_unscaled(2) = k_unscaled(2) * 3600;
    %k_unscaled(3) = k_unscaled(3) * 3600;
    %k_unscaled(4) = k_unscaled(4) * 3600;
    %k_unscaled(5) = k_unscaled(5) * 3600;
    %k_unscaled(6) = k_unscaled(6) * 3600 * 10^-12;
    k_unscaled(1) = k(1);
    k_unscaled(2) = k(2);
    k_unscaled(3) = k(3);
    k_unscaled(4) = k(4);
    k_unscaled(5) = k(5);
    k_unscaled(6) = k(6);
    num = (k_unscaled(1) * k_unscaled(3) * k_unscaled(5) * S
    - k_unscaled(2) * k_unscaled(4) * k_unscaled(6) * P) * Et;
    den1 = k_unscaled(2) * k_unscaled(5) + k_unscaled(2) * k_unscaled(4)
    + k_unscaled(3) * k_unscaled(5);
    den2 = S * k_unscaled(1) * (k_unscaled(3) + k_unscaled(4) + k_unscaled(5));
    den3 = P * k_unscaled(6) * (k_unscaled(2) + k_unscaled(3) + k_unscaled(4));
    velocity = num / (den1 + den2 + den3);
end
```

A.9 GMA Reaction

```
function velocity = Reaction_GMA(k, A, B, numSvar)
%Two Substrate GMA
g = zeros(2,1);
% if(k(1) <= .33333)
%     kf = k(1)/.33333 * 1e1;
% elseif(k(1) <= .66666)
% kf = (k(1)-.33333)/.33333 * 1e2;
% elseif(k(1) <= .99999)
```

```
% kf = (k(1) - .66666)/.33333 * 1e3;
%    else
% kf = 1e3;
%    end
% for i=1:2,
%        if(k(i + 1) <= .33333)
%     g(i) = k(i+1)/.33333 * 1.0;
% elseif(k(i +1) <= .66666)
% g(i) = (k(i +1)-.33333)/.33333 * 2.0;
% elseif(k(i +1) <= .99999)
% g(i) = (k(i +1) - .66666)/.33333 * 3.0;
%        else
% g(i) = 3.0;
%        end
%    end
kf = k(1);
g(1) = k(2);
%g(2) = k(3);
if (numSvar == 1)
if (A < 1.0e-8)
velocity = 0.0;
        else
velocity = kf * A ^ g(1);
        end
elseif (numSvar == 2)
if ((A < 1.0e-8) || (B < 1.0e-8))
velocity = 0.0;
else
velocity = kf * A ^ g(1) * B ^ g(2);
```

```
        end
    end
end
```

A.10 Reversible GMA Reaction

```
function velocity = Reaction_RevGMA(k, A, B, P, Q, numSvar, numPvar)
%Two Substrate Two Product Reversible GMA
g = zeros(4,1);
% if(k(1) <= .33333)
%     kf = k(1)/.33333 * 1e1;
% elseif(k(1) <= .66666)
% kf = (k(1)-.33333)/.33333 * 1e2;
% elseif(k(1) <= .99999)
% kf = (k(1) - .66666)/.33333 * 1e3;
%    else
% kf = 1e3;
%    end
% if(k(2) <= .33333)
%     kr = k(2)/.33333 * 1e1;
% elseif(k(2) <= .66666)
% kr = (k(2)-.33333)/.33333 * 1e2;
% elseif(k(2) <= .99999)
% kr = (k(2) - .66666)/.33333 * 1e3;
%    else
% kr = 1e3;
%    end
% for i=1:(numSvar + numPvar),
%        if(k(i + 1) <= .33333)
%     g(i) = k(i+1)/.33333 * 1.0;
```

```
% elseif(k(i +1) <= .66666)
% g(i) = (k(i +1)-.33333)/.33333 * 2.0;
% elseif(k(i +1) <= .99999)
% g(i) = (k(i +1) - .66666)/.33333 * 3.0;
%        else
% g(i) = 3.0;
%        end
%    end
kf = k(1);
kr = k(2);
g(1) = k(3);
g(2) = k(4);
if (numSvar == 1)
if (A < 1.0e-8)
velocityF = 0.0;
        else
velocityF = kf * A ^ g(1);
        end
elseif (numSvar == 2)
if ((A < 1.0e-8) || (B < 1.0e-8))
velocityF = 0.0;
else
velocityF = kf * A ^ g(1) * B ^ g(2);
        end
    end
if (numPvar == 1)
if (P < 1.0e-8)
velocityR = 0.0;
        else
```

```
velocityR = kr * P ^ g(numSvar + 1);
        end
elseif (numSvar == 2)
if ((P < 1.0e-8) || (Q < 1.0e-8))
velocityR = 0.0;
else
velocityR = kr * P ^ g(numSvar + 1) * Q ^ g(numSvar + 2);
        end
    end
    velocity = velocityF - velocityR;
end
```

APPENDIX B

INTEGRATOR TEST PROBLEMS IN MATLAB

This chapter contains the integrator test problems in their Matlab form.

B.1 OREGO

```
function dydt = OREGO_Model(t,y)

dydt = zeros(3,1);

dydt(1) = 77.27 * (y(2) + y(1) * (1.0 - 8.375e-5 * y(1) - y(2)));

dydt(2) = (y(3) - (1.0 + y(1)) * y(2)) / 77.27;

dydt(3) = 0.1610 * (y(1) - y(3));
```

B.2 HIRES

```
function dydt = HIRES_Model(t,y)

dydt = zeros(8,1);

dydt(1) = -1.71 * y(1) + 0.43 * y(2) + 8.32 * y(3) + 0.0007;

dydt(2) = 1.71 * y(1) - 8.75 * y(2);

dydt(3) = -10.03 * y(3) + .43 * y(4) + 0.035 * y(5);

dydt(4) = 8.32 * y(2) + 1.71 * y(3) - 1.12 * y(4);

dydt(5) = -1.745 * y(5) + 0.43 * y(6) + 0.43 * y(7);

dydt(6) = -280.0 * y(6) * y(8) + 0.69 * y(4) + 1.71 * y(5) - 0.43 * y(6)

+ 0.69 * y(7);

dydt(7) = 280.0 * y(6) * y(8) - 1.81 * y(7);

dydt(8) = -280.0 * y(6) * y(8) + 1.81 * y(7);
```

B.3 CLAUS

```
function dydt = CLAUS_Model(t,y)
```

```
dydt = zeros(10,1);
dydt(1) = 1.0 - 2.0 * y(1) + y(2);
dydt(2) = y(1) - 2.0 * y(2) + y(3);
dydt(3) = y(2) - 2.0 * y(3) + y(4);
dydt(4) = y(3) - 2.0 * y(4) + y(5);
dydt(5) = y(4) - 2.0 * y(5) + y(6);
dydt(6) = y(5) - 2.0 * y(6) + y(7);
dydt(7) = y(6) - 2.0 * y(7) + y(8);
dydt(8) = y(7) - 2.0 * y(8) + y(9);
dydt(9) = y(8) - 2.0 * y(9) + y(10);
dydt(10) = y(9) - 2.0 * y(10);
```

B.4 HOGK

```
function dydt = HODGK_Model(t,y)
dydt = zeros(4,1);
an = 0.01 * (y(4) + 10.0) / (exp(0.1 * y(4) + 1.0) - 1.0);
bn = 0.125 * exp(y(4) / 80.0);
am = 0.1 * (y(4) + 25.0) / (exp(0.1 * y(4) + 2.5) - 1.0);
bm = 4.0 * exp(y(4) / 18.0);
ah = 0.07 * exp(0.05 * y(4));
bh = 1.0 / (exp(0.1 * y(4) + 3.0) + 1.0);
dydt(1) = an * (1.0 - y(1)) - bn * y(1);
dydt(2) = am * (1.0 - y(2)) - bm * y(2);
dydt(3) = ah * (1.0 - y(3)) - bh * y(3);
dydt(4) = -36.0 * pow(y(1), 4) * (y(4) - 12.0) - 0.3 * (y(4) + 10.6) - 120.0
* pow(y(2), 3) * y(3) * (y(4) + 115.0);
```

B.5 PO

```
function dydt = Horseradish_Model(t,y)
```

```
%Parameters
k1 = 1;
k2 = 1250;
k3 = 0.046875;
k4 = 20;
k5 = 1.104;
k6 = 0.001;
k7 = 0.89;
kn7 = 0.1175;
k8 = 0.5;
N = zeros(4,9);
N(1,1) = -1;
N(2,1) = -1;
N(3,1) = 1;
N(3,2) = -2;
N(4,2) = 2;
N(1,3) = -1;
N(2,3) = -1;
N(3,3) = 2;
N(4,3) = -1;
N(3,4) = -1;
N(4,5) = -1;
N(3,6) = 1;
N(1,7) = 1;
N(2,8) = 1;
N(1,9) = -1;
V = zeros(9,1);
V(1) = k1 * y(1) * y(2) * y(3);
V(2) = k2 * y(3) ^ 2;
```

```
V(3) = k3 * y(1) * y(2) * y(4);

V(4) = k4 * y(3);

V(5) = k5 * y(4);

V(6) = k6;

V(7) = k7;

V(8) = k8;

V(9) = kn7 * y(1);

dydt = N * V;
```

B.6 PBPK

B.6.1 Model

```
function dydt = PBPK_Model(t,y)

Qc = .235 * .250 ^ .75;

dydt = zeros(length(y),1);

V(1) = 0.0272;

V(2) = 0.0544;

V(3) = 0.0366;

V(4) = 0.005;

V(5) = 0.076;

V(6) = 0.0057;

V(7) = 0.027;

V(8) = 0.002;

V(9) = 0.0033;

V(10) = 0.0073;

V(11) = 0.404;

V(12) = 0.19;

V(13) = 0.04148;

Q(1) = Qc;

Q(2) = Qc;
```

```
Q(3) = 0.175 * Qc;

Q(4) = Qc;

Q(5) = 0.07 * Qc;

Q(6) = 0.02 * Qc;

Q(7) = 0.131 * Qc;

Q(8) = 0.02 * Qc;

Q(9) = 0.049 * Qc;

Q(10) = 0.141 * Qc;

Q(11) = 0.278 * Qc;

Q(12) = 0.058 * Qc;

Q(13) = 0.122 * Qc;

Ptp(1) = 5.62;

Ptp(2) = 0;

Ptp(3) = 4.99;

Ptp(4) = 5.62;

Ptp(5) = 12.09;

Ptp(6) = 11.82;

Ptp(7) = 7.20;

Ptp(8) = 2.65;

Ptp(9) = 3.87;

Ptp(10) = 4.59;

Ptp(11) = 2.89;

Ptp(12) = 6.32;

Ptp(13) = 6.03;

% double BP = 1.04;

BP = 1.04;

Eh = 0.851;

%Simplification Variables

u = (Q(3) - Q(7) - Q(8)) * y(1);
```

```
v = Q(7) * y(7) / (Ptp(7)/BP) + Q(8) * y(8) / (Ptp(8)/BP);
dydt(1) = PBPK_Mass(Q(1), -y(1), -y(4), V(1), Ptp(1), BP);
dydt(2) = 0.0;
dydt(3) = (u + v - Q(3) * y(3) / (Ptp(7)/BP)) / V(3) - ((u + v) * Eh)/V(3);
dydt(4) = PBPK_Mass(Q(4), y(2), y(4), V(4), Ptp(4), BP);
dydt(2) = dydt(2) + PBPK_Mass2(Q(3), Q(2), y(3), y(2), V(2), Ptp(3), BP);
for i = 5:13,
    dydt(i) = PBPK_Mass(Q(i), y(1), y(i), V(i), Ptp(i), BP);
    dydt(2) = dydt(2) + PBPK_Mass2(Q(i), Q(2), y(i), y(2), V(2), Ptp(i), BP);
end
```

B.6.2 Supporting Functions

```
function dCdt = PBPK_Mass(Q, C1, C2, V, Ptp, BP)
Cvbt = C2 / (Ptp/BP);
dCdt = (Q * (C1 - Cvbt)) / V;


function dCdt = PBPK_Mass2(Q1, Q2, C1, C2, V, Ptp, BP)
Cvbt = C1 / (Ptp/BP);
dCdt = (Q1 * Cvbt - Q2 * C2) / V;
```

B.7 HER2

```
function dydt = HER2_Model(t,y)
dydt = zeros(18,1);
%Model Parameters
kon = 9.7 * 10^7;
koff = 0.24;
konFab = 1.4 * 10^7;
koffFab = 0.30;
keR1 = 0.08;
keR2 = 0.03;
```

```
keR1R2 = 0.04;
keR1L = 0.28;
keR1R2L = 0.10;
kc = 1.0 * 10 ^ -3;
kuR1R1 = 10.0;
kuR1R2 = 10.0;
kuR2R2 = 1.0;
kuR1R2L = 0.1;
kuLR1R1L = 0.1;
L = 16.0 * 10 ^ -9;
Ab = 1.3 * 10 ^ -9;
dydt(1) = -kc * y(1) * y(2) + kuR1R2 * y(4) - 2 * kc * y(1) * y(1) +
2 * kuR1R1 * y(3) - kc * y(1) * y(10) + kuR1R2 * y(11) - kon * L * y(1)
+ koff * y(6) - kc * y(1) * y(6) + kuR1R1 * y(8);
dydt(2) = -2 * kc * y(2) * y(2) + 2 * kuR2R2 * y(5) - kc * y(1) * y(2) +
kuR1R2 * y(4) - konFab * y(2) * Ab + koffFab * y(10) - kc * y(2)* y(10)
+ kuR2R2 * y(12) - kc * y(6) * y(2) + kuR1R2L * y(7);
dydt(3) = kc * y(1) * y(1)- kuR1R1 * y(3) - kon * L * y(3) + koff * y(8);
dydt(4) = kc * y(1) * y(2) - kuR1R2 * y(4) - konFab * Ab * y(4)
+ koffFab * y(11) - kon * L * y(4) + koff * y(7);
dydt(5) = kc * y(2) * y(2) - kuR2R2 * y(5) - konFab * Ab * y(5) + koffFab  * y(12);
dydt(6) = kon * L * y(1) - koff * y(6) - 2 * kc * y(6) * y(6) + 2 * kuLR1R1L * y(9)
- kc * y(6) * y(2) + kuR1R2L * y(7) - kc * y(6) * y(10) + kuR1R2L * y(14)
- kc * y(1) * y(6) + kuR1R1 * y(8) - keR1L * y(6);
dydt(7) = kon * L * y(4) - koff * y(7) + kc * y(6) * y(2) - kuR1R2L * y(7)
- konFab * Ab * y(7) + koffFab * y(14) - keR1R2L * y(7);
dydt(8) = kon * L * y(3) - koff * y(8) - kon * L * y(8) + koff * y(9)
+ kc * y(1) * y(6) - kuR1R1 * y(8) - keR1L * y(8);
dydt(9) = kc * y(6) * y(6) - kuLR1R1L * y(9) + kon * L * y(8)
```

```
- koff * y(9) - keR1L * y(9);
dydt(10) = konFab * y(2) * Ab - koffFab * y(10) - 2 * kc * y(10) * y(10)
+ 2 * kuR2R2 * y(13) - kc * y(1) * y(10) + kuR1R2 * y(11) - kc * y(6) * y(10)
- kc * y(2) * y(10) + kuR2R2 * y(12) - keR2 * y(10);
dydt(11) = kc * y(1) * y(10) - kuR1R2 * y(11) + konFab * Ab * y(4) - koffFab * y(11)
- kon * L * y(11)  + koff * y(14) - keR1R2 * y(11);
dydt(12) = konFab * Ab * y(5) - koffFab * y(12) - konFab * Ab * y(12)
+ koffFab * y(13) + kc * y(2) * y(10) - kuR2R2 * y(12) - keR2 * y(12);
dydt(13) = konFab * Ab * y(12) - koffFab * y(13) + kc * y(10) * y(10)
- kuR2R2 * y(13) - keR2 * y(13);
dydt(14) = kc  * y(6) * y(10) - kuR1R2L * y(14) + kon * L * y(11) - koff * y(14)
+ konFab * Ab * y(7) - koffFab * y(14) - keR1R2L * y(14);
dydt(15) = keR1L * y(6) + keR1L * y(8) + keR1R2L * y(7) + 2 * keR1L * y(9);
dydt(16) = keR2 * y(10) + keR2 * y(12) + 2 * keR2 * y(13)
+ keR1R2 * y(11) + keR1R2L * y(14);
dydt(17) = keR1L * y(6) + 2 * keR1L * y(9) + keR1R2L * y(7) + 2 * keR1L * y(8)
+ keR1R2L * y(14) + keR1 * y(11);
dydt(18) = keR1R2L * y(7) + keR2 * y(10) + 2 * keR2 * y(13) + 2 * keR2 * y(12)
+ keR1R2 * y(11) + keR1R2L * y(14);
```

B.8 SHINGO

```
function dydx = SPHINGO_Model(t,y)
X26 = .266e-2;
X27 = .262e-3;
X28 = 1100;
X29 = .54e-5;
X30 = .508e-1;
X31 = .13e-2;
X32 = .45e-2;
```

```
X33 = .33e-3;

X34 = .165e-4;

X35 = .1650000000e-3;

X36 = .4e-5;

X37 = 446;

X38 = .332e-2;

X39 = .24e-2;

X40 = .61e-3;

X41 = .8e-3;

X42 = .66e-3;

X43 = .1e-3;

X44 = .172e-2;

X45 = .1e-2;

X46 = .833e-3;

X47 = 1176;

X48 = 20;

X49 = .394e-2;

X50 = .367e-4;

X51 = .15e-3;

X52 = .89e-2;

X53 = .198e-4;

X54 = .17e-3;

X55 = .8250000000e-4;

X56 = .1066e-4;

X57 = .106e-3;

X58 = .5e-1;

X59 = .6000000000e-3;

X60 = .22e-1;

X61 = 60;
```

```
X62 = .1e-1;
X63 = .73;
X64 = .15e-3;
dydx = zeros(25,1);
dydx(1) = 174059.9223* X57 * y(12)^.9986438167 * y(13)^.1980849053
- 671635.4946 * y(1)^.9996667780 * X27;
dydx(2) = 671635.4946 * y(1)^.9996667780 * X27
+ 3057.424256 * y(3)^.5000000001 * X29 + 12.01395274 * y(4)^.9688581315 * X41
- 196097.2998 * y(2)^.9642857143 * X34 * y(23)^ .5278118802
- 2626.951478 * y(2)^.9743589746 * X28^.1120348511e-2 * X36
- 50469.00266 * y(2)^.5000000000 * X54;
dydx(3) = 196097.2998 * y(2)^.9642857143 * X34 * y(23)^.5278118802
+ 89.10166634 * y(8)^.9722222223 * X64 + 152.5467333 * X64 * y(18)^.9296482412
+ 5371.916271 * X64 * y(19)^.9955924294 - 3057.424256 * y(3)^.5000000001 * X29
- 549.5746692 * X54 * y(3)^.4999999998
- 10.85002492 * X33* y(15)^1.685 * y(2)^-.3358742751e-2
* y(5)^-.2424327076e-1 * y(3)^.9739478958;
dydx(4) = 2626.951478 * y(2)^.9743589746 * X28^.1120348511e-2 * X36
- 12.01395274 * y(4)^.9688581315 * X41 - 2224.471846 * y(4)^.9604829853 * X50;
dydx(5) = 50469.00266 * y(2)^.5000000000 * X54
+ 222.9333962 * X41 * y(6)^.8615384613 + 2547.853547 * y(7)^.5000000002 * X53
- 818569.3952 * y(5)^.8000000000* X34 * y(23)^.5278118802
- 81726.07870 * y(5)^.9600000000 * X28^.1120348511e-2 * X36;
dydx(6) = 81726.07870 * y(5)^.9600000000 * X28^.1120348511e-2 * X36
-222.9333962 * X41 * y(6)^.8615384613 - 36885.26610 * y(6)^.8293838859 * X50;
dydx(7) = 818569.3952 * y(5)^.8000000000 * X34 * y(23)^.5278118802
+ 297.0055545 * y(8)^.9722222223 * X51 + 549.5746692 * X54 * y(3)^.4999999998
+ 355.9423777 * X51 * y(18)^.9296482412 + 8694.450406 * X51 * y(19)^.9955924294
- 2547.853547 * y(7)^.5000000002 * X53
```

```
- 10.49201130 * X33 * y(15)^1.685 * y(7)^.9629101285 * y(2)^ -.3320804470e-2
* y(5)^ -.2397510852e-1 - 37651.04526 * y(7)^.4999999997 * X43;
dydx(8) = 10.49201130 * X33 * y(15)^1.685 * y(7)^.9629101285
* y(2)^-.003320804470 * y(5)^-.02397510852 + 10.85002492 * X33 * y(15)^1.685
* y(2)^-.003358742751 * y(5)^.02424327076 * y(3)^.9739478958 + .1983043588e-1
* y(20)^.5 - 297.0055545 * y(8)^.9722222223 * X51
- 675.1417734 * y(8)^.5000000000 * X35
- 89.10166634 * y(8)^.9722222223 * X64 -.06262242910* y(8)^.5;
dydx(9) = 2508.655816 * y(11)^.9940357859 * X40
- 212.2543754 * y(13)^.0002063074440 * y(9)^.9933310571 * X38
* y(2)^-.006568638313 * y(5)^-.01305154574 * y(16)^-.2704826039
* y(14)^ -.01424888297 * y(11)^.235 * y(15)^.088 - 107.6712152
* y(9)^.9122962616 * y(16)^.06424469194 * X26;
dydx(10) = 212.2543754 * y(13)^.0002063074440 * y(9)^.9933310571 * X38
* y(2)^-.006568638313 * y(5)^-.0130515457 * y(16)^ -.2704826039
* y(14)^ -.01424888297 * y(11)^.235 * y(15)^ .088
- 88659.84118 * y(10)^ .5307262571 *X56;
dydx(11) = 113966.8672 * y(12) * X49 * y(10)^ .1493
- 94.55645807 * y(11)^ .4230769231 * X39 * y(2)^ -.02071563088 * y(5)^-.05022831050
* y(9)^ .326 * y(15)^ .248 - 2508.655816 * y(11)^ .9940357859 *X40;
dydx(12) = 23067.02892 * X58^.9975062347 * X30
+ 476.8620195 * y(25)^.007910349154 * y(24)^.1318391563 * X52
+ 2224.471846 * y(4)^.9604829853 * X50 + 36885.26610 * y(6)^.8293838859 * X50
- 174059.9223 * X57 * y(12)^ .9986438167 * y(13)^ .1980849053
- 113966.8672 * y(12) * X49 * y(10)^ .1493 - 300.0628238 * y(12) * X48
- 16807.75082 * y(12)^.9999230829 * y(24)^ .4157339305 * X59;
dydx(13) = 2227.478731 * X37^.1663551402 * X31
- 212.2543754 * y(13)^.0002063074440 * y(9)^.9933310571 * X38
* y(2)^-.006568638313 * y(5)^ -.01305154574 * y(16)^ -.2704826039
```

```
* y(14)^ -.01424888297 * y(11)^ .235 * y(15)^.088 - 174059.9223 * X57
* y(12)^.9986438167 * y(13)^.1980849053 - 192.8863284 * y(13)^.1999999997 * X32;
dydx(14) = 94.55645807 * y(11) ^ .4230769231 * X39 * y(2)^ -.02071563088
* y(5)^ -.05022831050 * y(9)^ .326 * y(15)^ .248 + 10.49201130 * X33
* y(15)^1.685 * y(7)^.9629101285 * y(2)^ -.003320804470 * y(5)^ -.02397510852
+ 10.85002492 * X33 * y(15)^1.685 * y(2)^ -.003358742751 * y(5)^ -.02424327076
* y(3)^ .9739478958 + 6.574294837 * y(15)^1.685 * y(18)^ .5000000001 * X55
- 31.01460421 * X45 * y(14)^.9683760683 * y(17)^ .5000000003
- 22634.48184 * X42 * y(14)^.9867081784;
dydx(15) = 107.6712152 * y(9)^ .9122962616 * y(16)^ .06424469194 * X26
- 10.49201130 * X33 * y(15)^1.685 * y(7)^ .9629101285 * y(2)^ -.003320804470
* y(5)^ -.02397510852 - 10.85002492 * X33 * y(15)^ 1.685 * y(2)^ -.003358742751
* y(5)^ -.02424327076 * y(3)^ .9739478958 - 62.50743872 * y(15)^.9441006585
* X28^.3344404739e-2 * X44 - 6.574294837 * y(15)^1.685 * y(18)^ .5000000001 * X55;
dydx(16) = 56.94734470 * X46 * X47^.5008488966
- 107.6712152 * y(9)^.9122962616 * y(16)^ .06424469194 * X26;
dydx(17) = 2224.471846 * y(4)^.9604829853 * X50
+ 36885.26610 * y(6)^ .8293838859 * X50
- 31.01460421 * X45 * y(14)^ .9683760683 * y(17)^.500000000;
dydx(18) = 675.1417734 * y(8)^ .5000000000 * X35
+ .01692654532 * y(21)^.5
- 152.5467333 * X64 * y(18)^ .9296482412
- 355.9423777 * X51 * y(18)^ .9296482412
- 6.574294837 * y(15)^ 1.685 * y(18)^ .5000000001 * X55
- .05345224838 * y(18)^.5;
dydx(19) = 6.574294837 * y(15)^ 1.685 * y(18)^ .5000000001 * X55
+ .01807753815 * y(22)^ .5
- 5371.916271 * X64 * y(19)^ .9955924294
- 8694.450406 * X51 * y(19)^ .9955924294 -.1084652289 * y(19)^ .5;
```

```
dydx(20) = .06262242910* y(8)^ .5 -.01983043588 * y(20)^.5;

dydx(21) = .5345224838 * y(18)^.5 - .01692654532 * y(21)^.5;

dydx(22) = .1084652289 * y(19)^.5 - .01807753815 * y(22)^.5;

dydx(23) = 16807.75082 * y(12)^.9999230829 * y(24)^.4157339305 * X59
- 196097.2998 * y(2)^.9642857143 * X34 * y(23)^ .5278118802
- 818569.3952 * y(5)^ .8000000000 * X34 * y(23)^ .5278118802;

dydx(24) = 17.25388687 * y(25)^ .04460212912
* y(12)^ -.04577407228 * y(23)^ -.1583280558 * X28^.3750000000 * X60
- 16807.75082 * y(12)^ .9999230829 * y(24)^ .4157339305 * X59
- 476.8620195 * y(25)^ .007910349154 * y(24)^ .1318391563 * X52;

dydx(25) = 1.757713590 * X28^.5000000001 * X62^.9999519254
* X63 * X61^.7986577182 * y(12)^ -.111
- 17.25388687 * y(25)^.04460212912 * y(12)^ -.04577407228
* y(23)^ -.1583280558 * X28 ^.3750000000 * X60;
```

APPENDIX C

HIGH PERFOMANCE COMPUTING MONTE CARLO C SOURCE

This chapter contains a complete C/MPI Monte Carlo Code.

C.1 Main

```
//Peter Henning 2005
//Monte Carlo MPI Estimation Code
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <string.h>
#include <mpi.h>
#define NEQ 6
#define NCN 12
#define NTSPAN 7
#define NUMVAR 12
int MonteCarloDriver(int iter, long double k[], long double X[], int F[], int R[],
long double Et[], int numVar, int neq, int ncn, int timepts, long seed,
char save_root[], int id, int p);
long double CostFunction(long double k[], long double X[], int F[], int R[],
long double Et[], int neq, int ncn, int RowsX, int ColsX, long double timespan[],
long double Y_Zero[]);
int ShampineStiffInetgrator(long double tout[], long double yout[],
long double timespan[], long double Y_Zero[], long double k[], int F[], int R[],
long double Et[], int neq, int ncn, int ntspan);
```

```
int OdeConstructor(long double t, long double y[], long double ydot[],
long double k[], int F[], int R[], long double Et[], int neq, int ncn);
int JacobianConstructor(long double t,long double y[],long double dfdy[],
long double k[], int F[],int R[],long double Et[],int neq, int ncn);
int MaxVectorScalerCompare (long double wt[], long double y[],
long double threshold, int neq);
int MaxVectorVectorCompare (long double temp[], long double y[],
long double ynew[], int neq);
long double InfinityNormAndDivide (long double yp[], long double wt[], int neq);
long double InfinityNormAndMultiply (long double ynew[], long double invwt[],
int neq);
long double MinScalerScalerCompare (long double hmax, long double htspan);
long double MaxScalerScalerCompare (long double absh, long double hmin);
int SquareMaxtrixTimesVector(long double tempvector[], long double dfdy[],
long double yp[], int neq);
int LUDecomp (long double Miter[], int neq, int indx[]);
int LUSolver(long double Miter[], int neq, int indx[], long double del[]);
int Cumprod (long double difRU[], int n);
int MatrixMultiplySpecial (long double difRU[], long double tempmatrix[],
long double difU[], int n, int m, int p, int rows_out, int rows_in1, int rows_in2);
int MatrixTimesVector(long double psi[], long double dif[],
long double tempvector2[], int rows, int cols, int row_size);
int SumRows (long double yinterp[], long double dif[], int rows,
int cols, int row_size);
int Interpolation (long double tempvector[], long double tinterp, long double tnew,
long double ynew[], long double h, long double dif[], int kcount, int neq);
long double Reaction_ZeroOrder(long double k[], long double Et);
long double Reaction_MassAction(long double k[], long double A, long double B,
int numSvar);
```

```
long double Reaction_RevMassAction(long double k[], long double A, long double B,
long double P, long double Q, int numSvar, int numPvar);
long double Reaction_MichaelisM(long double k[], long double S, long double Et);
long double Reaction_RevMichaelisM(long double k[], long double S, long double P,
long double Et);
long double Reaction_UniUni(long double k[], long double S, long double P,
long double Et);
long double Reaction_GMA(long double k[], long double A, long double B,
int numSvar);
long double Reaction_RevGMA(long double k[], long double A, long double B,
long double P, long double Q, int numSvar, int numPvar);
long double kinetic_MassActionSder(long double k[], long double A,
long double B, int numSvar);
long double kinetic_RevMassActionSder(long double k[], long double A, long double B,
long double P, long double Q, int numSvar, int numPvar);
long double kinetic_RevMassActionPder(long double k[], long double A, long double B,
long double P, long double Q, int numSvar, int numPvar);
long double kinetic_MichaelisSder(long double k[], long double S, long double Et);
long double kinetic_RevMichaelisSder(long double k[], long double S,
long double P, long double Et);
long double kinetic_RevMichaelisPder(long double k[], long double S,
long double P, long double Et);
long double kinetic_uniunisder(long double k[],long double S,
long double P, long double Et);
long double kinetic_uniunipder(long double k[],long double S,
long double P,long double Et);
long double kinetic_GMASder(long double k[], long double A,
long double B, int numSvar);
long double kinetic_RevGMASder(long double k[], long double A, long double B,
```

```
long double P, long double Q, int numSvar, int numPvar);
long double kinetic_RevGMAPder(long double k[], long double A, long double B,
long double P, long double Q, int numSvar, int numPvar);
long double ran2(long *idum);
int main (int argc, char *argv[])
{
int ncn = NCN;
int neq = NEQ;
int ntspan = NTSPAN;
int numVar = NUMVAR;
long double X[NTSPAN * (NEQ+1)];
int F[NEQ * NCN];
int R[NCN];
long double Et[NCN];
long double k[NUMVAR];
long seed = -110605;
int i;
int j;
int id;
int p;
char save_root[30] = "MC_TEST";
MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &id);
MPI_Comm_size (MPI_COMM_WORLD, &p);
if (id == 0){
FILE *pRead;
pRead = fopen("F.txt", "r");
if (pRead == NULL) {
printf("\nFile cannot be Opened\n");
```

```
} else {
while ( !feof(pRead) ) {
for(i=0; i<neq; i++) {
for(j=0; j<ncn; j++) {
fscanf(pRead, "%d", &F[ncn * i + j]);
}
}
}
}
fclose(pRead);
pRead = fopen("XC16.txt", "r");
if (pRead == NULL) {
printf("\nFile cannot be Opened\n");
} else {
while ( !feof(pRead) ) {
for(i=0; i<ntspan; i++) {
for(j=0; j<(neq+1); j++) {
fscanf(pRead, "%Le", &X[(neq+1) * i + j]);
}
}
}
}
fclose(pRead);
//Initializing R and Et (Maybe from Files in the Future)
for(i=0; i<ncn; i++){
Et[i] = 1.0;
R[i] = 0;
}
R[0] = 0;
```

```
R[1] = 1;
R[2] = 1;
R[3] = 1;
R[4] = 1;
R[5] = 1;
R[6] = 1;
R[7] = 1;
R[8] = 1;
R[9] = 1;
R[10] = 1;
R[11] = 1;
Et[0] = 10000.0;
}
//Broadcast Variables from Master
MPI_Bcast (F, (NEQ * NCN), MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast (R, NCN, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast (Et, NCN, MPI_LONG_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast (X, (NTSPAN * (NEQ +1)), MPI_LONG_DOUBLE, 0, MPI_COMM_WORLD);
//Variable Initialization Complete
//Call Optimization Routine
int iter = 100;
MonteCarloDriver(iter, k, X, F, R, Et, numVar, neq, ncn, ntspan, seed,
save_root, id, p);
MPI_Finalize();
return 0;
}
```

C.2 Monte Carlo Driver

```
int MonteCarloDriver(int iter, long double k[], long double X[], int F[], int R[],
```

```
long double Et[], int numVar, int neq, int ncn, int timepts, long seed,
char save_root[], int id, int p)
{
int i;
int j;
int RowsX = timepts;
int ColsX = neq + 1;
long double timespan[NTSPAN];
long double Y_Zero[NEQ];
long double Cost;
long double BestCost = 16.0;
MPI_Status status;
if (id == 0) {
struct timeval tp1, tp2;
int process_num;
int finished = p - 1;
gettimeofday(&tp1, NULL);
char errorFile[37] = "Error_";
char paramFile[37] = "Param_";
FILE *pWrite1;
FILE *pWrite2;
//sprintf(idname,"%d", id);
strncat(errorFile, save_root, 36 - strlen(errorFile));
strncat(paramFile, save_root, 36 - strlen(paramFile));
//strncat(errorFile, idname, 36 - strlen(errorFile));
//strncat(paramFile, idname, 36 - strlen(paramFile));
pWrite1 = fopen(errorFile, "a");
pWrite2 = fopen(paramFile, "a");
while (finished != 0){
```

```
MPI_Recv(&process_num, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
MPI_Recv(&Cost, 1, MPI_LONG_DOUBLE, process_num, 0, MPI_COMM_WORLD, &status);
MPI_Recv(k, NUMVAR, MPI_LONG_DOUBLE, process_num, 0, MPI_COMM_WORLD, &status);
MPI_Recv(&i, 1, MPI_INT, process_num, 0, MPI_COMM_WORLD, &status);
//printf("I'm recieved message on iter %d\n", i);
if (Cost < BestCost) {
fprintf(pWrite1, "%Le\n", Cost);
for(j=0; j<numVar; j++){
fprintf(pWrite2, "%Lf\t", k[j]);
}
fprintf(pWrite2, "\n");
BestCost = Cost;
} else if (Cost < 5.3) {
fprintf(pWrite1, "%Le\n", Cost);
for(j=0; j<numVar; j++){
fprintf(pWrite2, "%Lf\t", k[j]);
}
fprintf(pWrite2, "\n");
}
if (i == iter){
finished = finished - 1;
}
}
fclose(pWrite1);
fclose(pWrite2);
gettimeofday(&tp2, NULL);
printf("%d\n", (int)(tp2.tv_sec - tp1.tv_sec));
} else {
seed = seed - id;
```

```
for(i=0; i<timepts; i++){
timespan[i] = X[i * ColsX];
}
for(i=0; i<neq; i++){
Y_Zero[i] = X[i+1];
}
for(i=0;i<iter;i++){
for(j=0; j<numVar; j++){
k[j] = ran2(&seed);
//k[j] = 0.5;
}
Cost = CostFunction(k, X, F, R, Et, neq, ncn, RowsX, ColsX, timespan, Y_Zero);
if ((Cost < BestCost) && (i != iter)){
//printf("I'm sending message on iter %d\n", i);
MPI_Send(&id, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&Cost, 1, MPI_LONG_DOUBLE, 0, 0, MPI_COMM_WORLD);
MPI_Send(k, NUMVAR, MPI_LONG_DOUBLE, 0, 0, MPI_COMM_WORLD);
MPI_Send(&i, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
BestCost = Cost;
} else if ((Cost < 5.3) && (i != iter)){
MPI_Send(&id, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&Cost, 1, MPI_LONG_DOUBLE, 0, 0, MPI_COMM_WORLD);
MPI_Send(k, NUMVAR, MPI_LONG_DOUBLE, 0, 0, MPI_COMM_WORLD);
MPI_Send(&i, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
}
printf("I'm sending message on iter %d\n", i);
MPI_Send(&id, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&Cost, 1, MPI_LONG_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

```
MPI_Send(k, NUMVAR, MPI_LONG_DOUBLE, 0, 0, MPI_COMM_WORLD);
MPI_Send(&i, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
return 0;
}
```

C.3 Cost Function

```
long double CostFunction(long double k[], long double X[], int F[], int R[],
long double Et[], int neq, int ncn, int RowsX, int ColsX,
long double timespan[], long double Y_Zero[])
{
int errcode = 1;
int i;
int j;
long double cost = 0.0;
long double tout[NTSPAN];
long double yout[NTSPAN * NEQ];
errcode = ShampineStiffInetgrator(tout, yout, timespan, Y_Zero, k, F, R,
Et, neq, ncn, RowsX);
if(errcode == 1){
cost = 15.0;
return cost;
}
if(errcode == 2){
cost = 9.0;
return cost;
}
if(errcode == 3){
cost = 10.0;
```

```
return cost;
}
for(i=0; i<neq; i++){
for (j=1; j<RowsX; j++){
cost += pow((X[j * ColsX + (i+1)] - yout[j * neq + i]), 2.0);
}
}
//printf("Cost is %Le\n", cost);
return log10(cost);
}
```

C.4 NDF Integrator

```
int ShampineStiffInetgrator(long double tout[], long double yout[],
long double timespan[], long double Y_Zero[], long double k[], int F[],
int R[], long double Et[], int neq, int ncn, int ntspan)
{
//Routine Initialization Steps
//Replacement for odearguments
int i;
int j;
int next = 1;
int tdir = 1;
long double t0 = timespan[0];
long double tfinal = timespan[NTSPAN-1];
long double f0[NEQ];
long double rtol = 1e-3;
long double threshold = 1.0e-3;
long double htspan = timespan[1] - timespan[0];
long double hmax = timespan[NTSPAN-1] * .10;
```

```
//Getting F_Zero
OdeConstructor(t0, Y_Zero, f0, k, F, R, Et, neq, ncn);
int Mt[NEQ][NEQ];
long double t = t0;
long double y[NEQ];
int maxk = 4;
for(i=0; i<neq; i++){
for(j=0; j<neq; j++){
if (i == j){
Mt[i][j] = 1;
} else {
Mt[i][j] = 0;
}
}
y[i] = Y_Zero[i];
}
//Initializing Method Parameters
long double G[5] = {1.0, 3.0/2.0, 11.0/6.0, 25.0/12.0, 137.0/60.0};
long double alpha[5] = {-37.0/200.0, -1.0/9.0, -0.0823, -0.0415, 0.0};
long double invGa[5];
long double erconst[5];
int kJ[5][5];
int kI[5][5];
long double difU[25] = {-1.0, -2.0, -3.0, -4.0, -5.0, 0.0, 1.0, 3.0, 6.0, 10.0, 0.0,
0.0, -1.0, -4.0, -10.0, 0.0, 0.0, 0.0, 1.0, 5.0, 0.0, 0.0, 0.0, 0.0, -1.0};
for(i=0;i<5; i++){
invGa[i] = 1.0 / (G[i] * (1 - alpha[i]));
erconst[i] = alpha[i] * G[i] + (1.0 / (i+2.0));
for(j=0; j<5; j++){
```

```
kI[i][j] = i+1;
kJ[i][j] = j+1;
}
}
int maxit = 3;
//Get Initial slope yp
long double yp[NEQ];
for(i=0; i<neq; i++){
yp[i] = f0[i];
}
//First Jacobian Call
long double dfdy[NEQ * NEQ];
int Jcurrent = JacobianConstructor(t, y, dfdy, k, F, R, Et, neq, ncn);
// hmin is a small number such that t + hmin is clearly different from t in
// the working precision, but with this definition, it is 0 if t = 0.
long double eps = 2.220446049250313e-16;
long double hmin = 16 * eps * fabs(t);
//Compute an initial step size h using yp = y'(t).
long double wt[NEQ];
MaxVectorScalerCompare(wt, y, threshold, neq);
long double rh = 1.25 * InfinityNormAndDivide(yp, wt, neq) / sqrt(rtol);
long double absh = MinScalerScalerCompare(hmax,htspan);
if ((absh * rh) > 1){
absh = 1 / rh;
}
absh = MaxScalerScalerCompare(absh, hmin);
//The error of BDF1 is 0.5*h^2*y''(t), so we can determine the optimal h.
long double h = tdir * absh;
long double tdel =(t + tdir * MinScalerScalerCompare(sqrt(eps) *
```

```
MaxScalerScalerCompare(fabs(t), fabs(t+h)), absh)) - t;
//Calculating f1
long double f1[NEQ];
OdeConstructor(t + tdel, y, f1, k, F, R, Et, neq, ncn);
long double dfdt[NEQ];
long double tempvector[NEQ];
SquareMaxtrixTimesVector(tempvector, dfdy, yp, neq);
for (i=0; i<neq; i++){
dfdt[i] = (f1[i] - f0[i])/tdel;
tempvector[i] = dfdt[i] + tempvector[i];
}
rh = 1.25 * sqrt(0.5 * InfinityNormAndDivide(tempvector, wt, neq) /rtol);
absh = MinScalerScalerCompare(hmax,htspan);
if ((absh * rh) > 1){
absh = 1 / rh;
}
absh = MaxScalerScalerCompare(absh, hmin);
h = tdir * absh;
//Initialize
int kcount = 0;
int klast = kcount;
long double abshlast = absh;
long double dif[NEQ * (4+3)];
for (i=0; i<neq; i++){
for (j=0; j<=maxk+2; j++){
if (j == 0){
dif[i * 7 + j] = h * yp[i];
} else {
dif[i * 7 + j] = 0.0;
```

```
}
}
}
long double hinvGak = h * invGa[kcount];
int nconhk = 0;
long double Miter[NEQ * NEQ];
for (i=0; i<neq; i++){
for (j=0; j<neq; j++){
Miter[i * neq + j] = Mt[i][j] - hinvGak * dfdy[i * neq + j];
}
}
int indx[NEQ];
LUDecomp(Miter, neq, indx);
int havrate = 0;
//Allocate memory for generated output
int nout = 0;
tout[nout] = t;
for(i=0; i<neq; i++){
yout[nout * neq + i] = y[i];
}
//The Main Loop
int done = 0;
long double tempMatrix1[25];
long double tempMatrix2[NEQ * (4+2)];
long double difRU[25];
int nofailed = 1;
int gotynew = 0;
long double psi[NEQ];
long double tempvector2[5];
```

```
long double tnew;
long double pred[NEQ];
long double ynew[NEQ];
long double difkp1[NEQ];
long double invwt[NEQ];
long double minnrm = 0.0;
int tooslow = 0;
long double rhs[NEQ];
long double del[NEQ];
long double newnrm;
long double rate;
long double errit;
long double oldnrm;
long double err;
long double hopt;
long double errkm1;
long double hkm1;
int nsteps = 0;
int fsteps = 0;
int ssteps = 0;
long double tstep;
long double ystep[NEQ];
int oldnout;
long double temp;
long double temp2;
int kopt;
int madeMass = 0;
long double errkp1;
long double hkp1;
```

```
//Code Testing Variables
//int print = 1;
while (done == 0){
hmin = 16 * eps * fabs(t);
absh = MinScalerScalerCompare(hmax, MaxScalerScalerCompare(absh, hmin));
h = tdir * absh;
//Stretch the step if within 10% of tfinal-t.
if ((1.1 * absh)  >= fabs(tfinal -t)){
h = tfinal - t;
absh = fabs(h);
done = 1;
}
if ((absh != abshlast) || (kcount != klast)){
for(i=0; i<5; i++){
for(j=0; j<5; j++){
tempMatrix1[i * 5 + j] = (kI[i][j] - 1 - kJ[i][j] * (absh/abshlast)) / (kI[i][j]);
}
}
Cumprod(tempMatrix1, 5);
MatrixMultiplySpecial(difRU, tempMatrix1, difU, 5, 5, 5, 5, 5, 5);
MatrixMultiplySpecial(tempMatrix2, dif, difRU, neq, (kcount+1), (kcount+1), 7, 7, 5);
for(i=0; i<neq; i++){
for(j=0; j<=kcount; j++){
dif[i * 7 + j] = tempMatrix2[i * 7 + j];
}
}
hinvGak = h * invGa[kcount];
nconhk = 0;
for (i=0; i<neq; i++){
```

```
for (j=0; j<neq; j++){
Miter[i * neq + j] = Mt[i][j] - hinvGak * dfdy[i * neq + j];
}
}
LUDecomp(Miter, neq, indx);
havrate = 0;
}
//LOOP FOR ADVANCING ONE STEP
nofailed = 1;
while (1){
gotynew = 0;
while (gotynew == 0){
//Compute the constant terms in the equation for ynew
for(i=0; i<=kcount; i++){
tempvector2[i] = G[i] * invGa[kcount];
}
MatrixTimesVector(psi, dif, tempvector2, neq, (kcount+1), 7);
//Predict a solution at t+h
tnew = t + h;
if (done == 1){
tnew = tfinal;
}
h = tnew - t;
SumRows (tempvector, dif, neq, kcount+1, 7);
for(i=0; i<neq; i++){
pred[i] = y[i] + tempvector[i];
ynew[i] = pred[i];
//if (ynew[i] < 0.0){
//printf("y became neqative\n");
```

```
//return 2;
//}
difkp1[i] = 0.0;
}
//The difference, difkp1, between pred and the final accepted
//ynew is equal to the backward difference of ynew of the order
//kcount+1.  Initialize to zero for the iteration to compute ynew.
MaxVectorVectorCompare (tempvector, y, ynew, neq);
MaxVectorScalerCompare(invwt, tempvector, threshold, neq);
for(i=0; i<neq; i++){
invwt[i] = 1 / invwt[i];
}
minnrm = 100 * eps * InfinityNormAndMultiply(ynew, invwt, neq);
//Iterate with simplified Newton method
tooslow = 0;
for(i=0; i<=maxit; i++){
OdeConstructor(tnew, ynew, tempvector, k, F, R, Et, neq, ncn);
for(j=0; j<neq; j++){
rhs[j] = (hinvGak * tempvector[j]) - (psi[j]+difkp1[j]);
del[j] = rhs[j];
}
LUSolver(Miter, neq, indx, del);
newnrm = InfinityNormAndMultiply(del, invwt, neq);
for(j=0; j<neq; j++){
difkp1[j] = difkp1[j] + del[j];
ynew[j] = pred[j] + difkp1[j];
}
if (newnrm <= minnrm){
gotynew = 1;
```

```
break;
} else if (i == 0) {
if (havrate == 1){
errit = newnrm * rate / (1 - rate);
if (errit <= 0.05 * rtol){
gotynew = 1;
break;
}
} else {
rate = 0.0;
}
} else if (newnrm > 0.9 * oldnrm) {
tooslow = 1;
break;
} else {
rate = MaxScalerScalerCompare((0.9 * rate), (newnrm / oldnrm));
havrate = 1;
errit = newnrm * rate / (1 - rate);
if (errit <= (0.5 * rtol)) {
gotynew = 1;
break;
} else if (i == maxit) {
tooslow = 1;
break;
} else if ((0.5 * rtol) < (errit * pow(rate, (maxit - i)))) {
tooslow = 1;
break;
}
}
```

```
oldnrm = newnrm;
}  //End of Newton Loop
if (tooslow == 1){
//Speed up the iteration by forming new linearization or reducing h.
ssteps = ssteps + 1;
if (Jcurrent == 0){
OdeConstructor(t, y, f0, k, F, R, Et, neq, ncn);
Jcurrent = JacobianConstructor(t, y, dfdy, k, F, R, Et, neq, ncn);
} else if (absh <= hmin) {
//printf("Failed to Meet Integration Tolerances with reducing stepsize below %Le at
time %Lf\n", hmin, t);
tout[0] = -99.0;
tout[ntspan - 1] = -99.0;
return 1;
} else {
abshlast = absh;
absh = MaxScalerScalerCompare((0.3 * absh), hmin);
h = tdir * absh;
done = 0;
for(i=0; i<5; i++){
for(j=0; j<5; j++){
tempMatrix1[i * 5 + j] = (kI[i][j] - 1 - kJ[i][j] * (absh/abshlast)) / (kI[i][j]);
}
}
Cumprod(tempMatrix1, 5);
MatrixMultiplySpecial(difRU, tempMatrix1, difU, 5, 5, 5, 5, 5, 5);
MatrixMultiplySpecial(tempMatrix2, dif, difRU, neq, (kcount+1), (kcount+1), 7, 7, 5);
for(i=0; i<neq; i++){
for(j=0; j<=kcount; j++){
```

```
dif[i * 7 + j] = tempMatrix2[i * 7 + j];
}
}
hinvGak = h * invGa[kcount];
nconhk = 0;
}
for (i=0; i<neq; i++){
for (j=0; j<neq; j++){
Miter[i * neq + j] = Mt[i][j] - hinvGak * dfdy[i * neq + j];
}
}
LUDecomp(Miter, neq, indx);
havrate = 0;
}
}
//End of While Loop for getting ynew
//difkp1 is now the backward difference of ynew of order kcount + 1
err = InfinityNormAndMultiply(difkp1, invwt, neq) * erconst[kcount];
if (err > rtol) {
//Failed Step
fsteps = fsteps + 1;
if (absh <= hmin){
//printf("Failed to Meet Integration Tolerances with reducing stepsize below %Le at
time %Lf\n", hmin, t);
tout[0] = -99.0;
tout[ntspan - 1] = -99.0;
return 1;
}
abshlast = absh;
```

```
if (nofailed == 1) {
nofailed = 0;
temp = (rtol/err);
temp2 = 1.0/(kcount + 2);
hopt = absh * MaxScalerScalerCompare(0.1, (0.833 * pow(temp, temp2)));  // 1/1.2
if (kcount > 0) {
for(i=0; i<neq; i++){
tempvector[i] = dif[i * 7 + kcount] + difkp1[i];
}
errkm1 = InfinityNormAndMultiply(tempvector, invwt, neq) * erconst[kcount-1];
temp = (rtol/errkm1);
temp2 = 1.0/(kcount + 1);
hkm1 = absh * MaxScalerScalerCompare(0.1, (0.769 * pow(temp, temp2)));  // 1/1.3
if (hkm1 > hopt){
hopt = MinScalerScalerCompare (absh, hkm1);
kcount = kcount - 1;
}
}
absh = MaxScalerScalerCompare(hmin, hopt);
} else {
absh = MaxScalerScalerCompare(hmin, 0.5 * absh);
}
h = tdir * absh;
if (absh < abshlast){
done = 0;
}
for(i=0; i<5; i++){
for(j=0; j<5; j++){
tempMatrix1[i * 5 + j] = (kI[i][j] - 1 - kJ[i][j] * (absh/abshlast)) / (kI[i][j]);
```

```
}
}
Cumprod(tempMatrix1, 5);
MatrixMultiplySpecial(difRU, tempMatrix1, difU, 5, 5, 5, 5, 5, 5);
MatrixMultiplySpecial(tempMatrix2, dif, difRU, neq, (kcount+1), (kcount+1), 7, 7, 5);
for(i=0; i<neq; i++){
for(j=0; j<=kcount; j++){
dif[i * 7 + j] = tempMatrix2[i * 7 + j];
}
}
hinvGak = h * invGa[kcount];
nconhk = 0;

for (i=0; i<neq; i++){
for (j=0; j<neq; j++){
Miter[i * neq + j] = Mt[i][j] - hinvGak * dfdy[i * neq + j];
}
}
LUDecomp(Miter, neq, indx);
havrate = 0;

} else {

break;
}
} //End while (true)
nsteps = nsteps + 1;
if (nsteps > 5000) {
return 3;
```

```
}
for (i=0; i<neq; i++){
dif[i * 7 + kcount + 2] = difkp1[i] - dif[i * 7 + kcount + 1];
dif[i * 7 + kcount + 1] = difkp1[i];
ystep[i] = ynew[i];
}
for (i=kcount; i>=0; i--){
for (j=0; j < neq; j++){
dif[j * 7 + i] = dif[j * 7 + i] + dif[j * 7 + i + 1];
}
}
tstep = tnew;
oldnout = nout;
while (next <= ntspan) {
if (tdir * (tnew - timespan[next]) < 0){
break;
} else if (tnew == timespan[next]) {
nout = nout + 1;
tout[nout] = tnew;
for (i=0; i<neq; i++){
yout[nout * neq + i] = ynew[i];
}
next = next + 1;
break;
}
nout = nout + 1;
tout[nout] = timespan[next];
Interpolation (tempvector, timespan[next], tstep, ystep, h, dif, kcount, neq);
for (i=0; i<neq; i++){
```

```
yout[nout * neq + i] = tempvector[i];
}
next = next + 1;
}
klast = kcount;
abshlast = absh;
if ((nconhk+1) < (maxk+3)) {
nconhk = nconhk + 1;
} else {
nconhk = maxk+3;
}
if (nconhk >= (kcount + 3)) {
temp = 1.2 * pow((err/rtol), (1.0/(kcount + 2)));
if (temp > 0.1) {
hopt = absh / temp;
} else {
hopt = 10.0 * absh;
}
kopt = kcount;
if (kcount > 0){
for (i=0; i<neq; i++){
tempvector[i] = dif[i * 7 + kcount];
}
errkm1 = InfinityNormAndMultiply(tempvector, invwt, neq) * erconst[kcount-1];
temp = 1.3 * pow((errkm1/rtol), (1.0/(kcount + 1)));
if (temp > 0.1) {
hkm1 = absh / temp;
} else {
hkm1 = 10.0 * absh;
```

```
}
if (hkm1 > hopt) {
hopt = hkm1;
kopt = kcount - 1;
}
}
if (kcount < maxk){
for (i=0; i<neq; i++){
tempvector[i] = dif[i * 7 + kcount + 2];
}
errkp1 = InfinityNormAndMultiply(tempvector, invwt, neq) * erconst[kcount+1];
temp = 1.4 * pow((errkp1/rtol), (1.0/(kcount + 3)));
if (temp > 0.1) {
hkp1 = absh / temp;
} else {
hkp1 = 10.0 * absh;
}
if (hkp1 > hopt) {
hopt = hkp1;
kopt = kcount + 1;
}
}
if (hopt > absh) {
absh = hopt;
if (kcount != kopt){
kcount = kopt;
}
}
}
```

```
//Advance Integration One Step
t = tnew;
for (i=0; i<neq; i++){
y[i] = ynew[i];
}
Jcurrent = 0;
} //End While Not Done
return 0;
}
```

C.5 ODE Constructor

```
int OdeConstructor(long double t, long double y[], long double ydot[],
long double k[], int F[], int R[], long double Et[], int neq, int ncn)
{
int i;
int j;
int kstart = 0;
long double A = 0.0;
long double B = 0.0;
long double P = 0.0;
long double Q = 0.0;
int Aset = 0;
int Pset = 0;
long double k_fun[6];
long double ReactionVelocity[NCN];
long double total;
int numSvar = 0;
int numPvar = 0;
for(i=0; i < ncn; i++)
```

```
{
    A = 0.0;
    Aset = 0;
    B = 0.0;
    P = 0.0;
    Pset = 0;
    Q = 0.0;
    numSvar = 0;
    numPvar = 0;
    for(j=0; j < neq; j++)
    {
        if((F[ncn * j + i] == -1) && (Aset == 0))
        {
            A = y[j];
            Aset = 1;
            numSvar = 1;
} else  if((F[ncn * j + i]== -1) && (Aset != 0)) {
            B = y[j];
            numSvar = 2;
}
        if((F[ncn * j + i] == 1) && (Pset == 0))
        {
            P = y[j];
            Pset = 1;
            numPvar = 1;
} else  if((F[ncn * j + i] == 1) && (Pset != 1)) {
     Q = y[j];
     numPvar = 2;
}
```

```
    }
if(R[i] == 0)
{
   for(j=0;j<1;j++)
{
k_fun[j]=k[kstart+j];
}
ReactionVelocity[i] = Reaction_ZeroOrder(k_fun, Et[i]);
kstart=kstart+1;
}
if(R[i] == 1)
{
   for(j=0;j<1;j++)
{
k_fun[j]=k[kstart+j];
}
ReactionVelocity[i] = Reaction_MassAction(k_fun, A, B, numSvar);
kstart=kstart+1;
}
if(R[i] == 2)
{
   for(j=0;j<2;j++)
{
k_fun[j]=k[kstart+j];
}
ReactionVelocity[i] = Reaction_RevMassAction(k_fun, A, B, P, Q, numSvar, numPvar);
kstart=kstart+2;
}
if(R[i] == 3)
```

```
{
   for(j=0;j<3;j++)
{
k_fun[j]=k[kstart+j];
}
ReactionVelocity[i] = Reaction_MichaelisM(k_fun, A, Et[i]);
kstart=kstart+3;
}
if(R[i] == 4)
{
   for(j=0;j<4;j++)
{
k_fun[j]=k[kstart+j];
}
ReactionVelocity[i] = Reaction_RevMichaelisM(k_fun, A, P, Et[i]);
kstart=kstart+4;
}
    if(R[i] == 5)
    {
        for(j=0;j<6;j++)
{
k_fun[j]=k[kstart+j];
}
ReactionVelocity[i] = Reaction_UniUni(k_fun, A, P, Et[i]);
kstart=kstart+6;
}
    if(R[i] == 6)
    {
        for(j=0;j<(numSvar + 1);j++)
```

```
{
k_fun[j]=k[kstart+j];
}
ReactionVelocity[i] = Reaction_GMA(k_fun, A, B, numSvar);
kstart=kstart + (numSvar + 1);
}
    if(R[i] == 7)
    {
        for(j=0;j<(numSvar + numPvar + 2); j++)
{
k_fun[j]=k[kstart+j];
}
ReactionVelocity[i] = Reaction_RevGMA(k_fun, A, B, P, Q, numSvar, numPvar);
kstart=kstart + numSvar + numPvar + 2;
}
}
for(j=0; j<neq; j++)
{
total=0.0;
for(i=0; i<ncn; i++)
{
total+= F[ncn * j + i] * ReactionVelocity[i];
}
ydot[j] = total;
}
return 0;
}
```

C.6 Jacobian Constructor

```
int JacobianConstructor(long double t,long double y[], long double dfdy[],
long double k[], int F[], int R[], long double Et[], int neq, int ncn)
{
int i;
int j;
int kstart = 0;
int Row_S;
int Row_P;
int Row_S_set;
int Row_P_set;
long double k_fun[6];
long double A;
long double B;
long double P;
long double Q;
int Aset = 0;
int Pset = 0;
long double substrateDer[NCN];
long double productDer[NCN];
long double jpieces[NEQ][NCN];
int numSvar = 0;
int numPvar = 0;
for(i=0;i<ncn;i++)
{
    A = 0.0;
    Aset = 0;
    B = 0.0;
    P = 0.0;
```

```
    Pset = 0;
    Q = 0.0;
    numSvar = 0;
    numPvar = 0;
    for(j=0; j < neq; j++)
    {
        if((F[ncn * j + i] == -1) && (Aset == 0))
        {
            A = y[j];
            Aset = 1;
            numSvar = 1;
} else if((F[ncn * j + i]== -1) && (Aset != 0)) {
    B = y[j];
            numSvar = 2;
}
        if((F[ncn * j + i] == 1) && (Pset == 0))
        {
            P = y[j];
            Pset = 1;
            numPvar = 1;
} else if((F[ncn * j + i] == 1) && (Pset != 1)) {
    Q = y[j];
    numPvar = 2;
}
           }
    if(R[i] == 0)
    {
        for(j=0;j<1;j++)
{
```

```
k_fun[j]=k[kstart+j];
}
substrateDer[i] = 0.0;
productDer[i] = 0.0;
kstart=kstart + 1;
}
    if(R[i] == 1)
    {
        for(j=0;j<1;j++)
{
k_fun[j]=k[kstart+j];
}
substrateDer[i] = kinetic_MassActionSder(k_fun, A, B, numSvar);
productDer[i] = 0.0;
kstart=kstart + 1;
}
    if(R[i] == 2)
    {
        for(j=0;j<2;j++)
{
k_fun[j]=k[kstart+j];
}
substrateDer[i] = kinetic_RevMassActionSder(k_fun, A, B, P, Q, numSvar, numPvar);
productDer[i] = kinetic_RevMassActionPder(k_fun, A, B, P, Q,numSvar, numPvar);
kstart=kstart + 2;
}
    if(R[i] == 3)
    {
        for(j=0;j<3;j++)
```

```
{
k_fun[j]=k[kstart+j];
}
substrateDer[i] = kinetic_MichaelisSder(k_fun,A, Et[i]);
productDer[i] = 0.0;
kstart=kstart + 3;
}
    if(R[i] == 4)
    {
        for(j=0;j<4;j++)
{
k_fun[j]=k[kstart+j];
}
substrateDer[i] = kinetic_RevMichaelisSder(k_fun,A,P, Et[i]);
productDer[i] = kinetic_RevMichaelisPder(k_fun,A,P, Et[i]);
kstart=kstart + 4;
}
    if(R[i] == 5)
    {
        for(j=0;j<6;j++)
{
k_fun[j]=k[kstart+j];
}
substrateDer[i] = kinetic_uniunisder(k_fun,A,P,Et[i]);
productDer[i] = kinetic_uniunipder(k_fun,A,P,Et[i]);
kstart=kstart + 6;
}
    if(R[i] == 6)
    {
```

```
        for(j=0;j<(numSvar + 1);j++)
{
k_fun[j]=k[kstart+j];
}
substrateDer[i] = kinetic_GMASder(k_fun, A, B, numSvar);
productDer[i] = 0.0;
kstart=kstart + (numSvar + 1);
}
    if(R[i] == 7)
    {
        for(j=0;j<(numSvar + numPvar + 2); j++)
{
k_fun[j]=k[kstart+j];
}
substrateDer[i] = kinetic_RevGMASder(k_fun, A, B, P, Q, numSvar, numPvar);
productDer[i] = kinetic_RevGMAPder(k_fun, A, B, P, Q, numSvar, numPvar);
kstart=kstart + numSvar + numPvar + 2;
}
}
for(i=0;i< (neq * neq);i++)
{
dfdy[i] = 0.0;
}
for(i=0; i<ncn; i++)
{
Row_S_set = 0;
Row_P_set = 0;
for(j=0; j<neq; j++)
{
```

```
jpieces[j][i]=0;
if(F[ncn * j + i] == -1)
{
jpieces[j][i]= -1 * substrateDer[i];
Row_S=j;
Row_S_set = 1;
}
if(F[ncn * j + i] == 1)
{
jpieces[j][i] = productDer[i];
Row_P=j;
Row_P_set = 1;
}
}
if ((Row_S_set == 1) && (Row_P_set == 1)){
dfdy[Row_S * neq + Row_P]= -1 * productDer[i];
dfdy[Row_P * neq + Row_S]= substrateDer[i];
}


}
for(i=0; i<neq; i++)
{
for(j=0; j<ncn; j++)
{
dfdy[i * neq + i] += jpieces[i][j];
}
}
return 1;
}
```

C.7 Matrix Manipulation Codes

```
int MaxVectorScalerCompare (long double wt[], long double y[],
long double threshold, int neq)
{
int i;
for(i=0; i<neq; i++){
wt[i] = threshold;
if ((fabs(y[i])) > threshold){
wt[i] = fabs(y[i]);
}
}
return 0;
}
int MaxVectorVectorCompare (long double temp[], long double y[],
long double ynew[], int neq)
{
int i;
for(i=0; i<neq; i++){
temp[i] = 0.0;
if (fabs(y[i]) >= fabs(ynew[i])){
temp[i] = fabs(y[i]);
} else {
temp[i] = fabs(ynew[i]);
}
}
return 0;
}
long double InfinityNormAndDivide (long double yp[],
long double wt[], int neq)
```

```
{
int i;
long double maximum = fabs(yp[0]/wt[0]);
for(i=1; i<neq; i++){
if (maximum < fabs(yp[i]/wt[i])){
maximum = fabs(yp[i]/wt[i]);
}
}
return maximum;
}
long double InfinityNormAndMultiply (long double ynew[],
long double invwt[], int neq)
{
int i;
long double maximum = fabs(ynew[0] * invwt[0]);
for(i=1; i<neq; i++){
if (maximum < fabs(ynew[i] * invwt[i])){
maximum = fabs(ynew[i] * invwt[i]);
}
}
return maximum;
}
long double MinScalerScalerCompare (long double hmax, long double htspan)
{
long double minimum;
if (hmax < htspan){
minimum = hmax;
} else {
minimum = htspan;
```

```
}
return minimum;
}
long double MaxScalerScalerCompare (long double absh, long double hmin)
{
long double maximum;
if (absh > hmin){
maximum = absh;
} else {
maximum = hmin;
}
return maximum;
}
int SquareMaxtrixTimesVector(long double tempvector[], long double dfdy[],
long double yp[], int neq)
{
int i;
int j;
for(i=0; i<neq; i++){
tempvector[i] = 0.0;
for (j=0; j<neq; j++){
tempvector[i] += dfdy[i * neq + j] * yp[j];
}
}
return 0;
}
int LUDecomp (long double Miter[], int neq, int indx[])
{
int i;
```

```
int j;
int k;
int imax;
long double big;
long double dum;
long double sum;
long double temp;
long double vv[NEQ];
for(i=0; i<neq; i++){
big = 0.0;
for(j=0; j<neq; j++){
temp = fabs(Miter[i * neq + j]);
if (temp > big){
big = temp;
}
}
if (big == 0.0){
printf("The Matrix sent into LU Decomposition is Singular\n");
}
vv[i] = 1.0/ big;
}
for(j=0; j<neq; j++){
for (i=0; i<j; i++){
sum = Miter[i * neq + j];
for (k=0; k<i; k++){
sum -= Miter[i * neq + k] * Miter[k * neq + j];
}
Miter[i * neq + j] = sum;
}
```

```
big = 0.0;
for (i=j; i<neq; i++){
sum = Miter[i * neq + j];
for(k=0; k<j; k++){
sum -= Miter[i * neq + k] * Miter[k * neq + j];
}
Miter[i * neq + j] = sum;
dum = vv[i] * fabs(sum);
if (dum >= big){
big = dum;
imax = i;
}
}
if (j != imax){
for(k=0; k<neq; k++){
dum = Miter[imax * neq + k];
Miter[imax * neq + k] = Miter[j * neq + k];
Miter[j * neq + k] = dum;
}
vv[imax] = vv[j];
}
indx[j]=imax;
if (Miter[j * neq + j] == 0.0){
Miter[j * neq + j] = 1.0e-20;
}
if (j != (neq-1)){
dum = 1.0/(Miter[j * neq + j]);
for(i=j+1; i<neq; i++){
Miter[i * neq + j] *= dum;
```

```
}
}
}
return 0;
}
int LUSolver(long double Miter[], int neq, int indx[], long double del[])
{
int i;
int ii = 0;
int ip;
int j;
long double sum;
for (i=0;i<neq;i++) {
ip=indx[i];
sum=del[ip];
del[ip]=del[i];
if (ii) {
for(j=ii;j<=i-1;j++){
sum -= Miter[i * neq + j] * del[j];
}
} else if (sum) {
ii=i;
}
del[i]=sum;
}
for (i=(neq-1); i>=0; i--){
sum=del[i];
for (j=i+1; j<neq; j++){
sum -= Miter[i * neq + j] * del[j];
```

```
}
del[i]= sum /Miter[i * neq + i];
}
return 0;
}
int Cumprod (long double difU[], int n)
{
int i;
int j;
int k;
for(i=1; i<n; i++){
for(j=0; j<n; j++){
difU[i * n + j] = difU[i * n + j] * difU[(i-1) * n + j];
}
}
return 0;
}
int MatrixMultiplySpecial (long double difRU[], long double tempmatrix[],
long double difU[], int n, int m, int p, int rows_out, int rows_in1, int rows_in2)
{
int i;
int j;
int k;

for(i=0; i<n; i++){
for(j=0; j<p; j++){
difRU[i * rows_out + j] = 0.0;
for(k=0; k<m; k++){
difRU[i * rows_out + j] += tempmatrix[i * rows_in1 + k] * difU[k * rows_in2 + j];
```

```
}
}
}

return 0;
}
int MatrixTimesVector(long double psi[], long double dif[],
long double tempvector2[], int rows, int cols, int row_size)
{
int i;
int j;
for(i=0; i<rows; i++){
psi[i] = 0.0;
for (j=0; j<cols; j++){
psi[i] += dif[i * row_size + j] * tempvector2[j];
}
}
return 0;
}
int SumRows (long double tempvector[], long double dif[], int rows, int cols,
int row_size)
{
int i;
int j;
for(i=0; i<rows; i++){
tempvector[i] = 0.0;
for(j=0; j<cols; j++){
tempvector[i] += dif[i * row_size + j];
}
```

```
}
return 0;
}
```

C.8 Interpolation Function

```
int Interpolation (long double yinterp[], long double tinterp, long double tnew,
long double ynew[], long double h, long double dif[], int kcount, int neq)
{
int i;
int j;
long double tempvector[kcount];
long double tempvector2[NEQ];
long double s = (tinterp - tnew) / h;
if (kcount == 0){
for (i=0; i<neq; i++){
yinterp[i] = ynew[i] + dif[i * 7] * s;
}
} else {
for (i=0; i<kcount; i++){
tempvector[i] = (s + i) / (i+1);
}
MatrixTimesVector(tempvector2, dif, tempvector, neq, (kcount+1), 7);
for (i=0; i<neq; i++){
yinterp[i] = ynew[i] + tempvector2[i];
}
}
return 0;
}
```

C.9 Zero Order Reaction

```
long double Reaction_ZeroOrder(long double kfun[], long double Et)
{
long double velocity;
velocity = kfun[0] * Et;
return velocity;
}
```

C.10 Mass Action Reaction

```
long double Reaction_MassAction(long double kfun[], long double A, long double B,
int numSvar)
{
long double k_scaled;
long double velocity;
if (kfun[0] <= .25){
k_scaled = kfun[0] / .25 * 1e-2;
} else if (kfun[0] <= .50){
k_scaled = (kfun[0] - .25) / .25 * 1e-1;
} else if (kfun[0] <= .75){
k_scaled = (kfun[0] - .50) / .25;
} else {
k_scaled = (kfun[0] - .75) / .25 * 1e1;
}
if (numSvar == 1){
velocity = k_scaled * A;
} else if (numSvar == 2) {
velocity = k_scaled * A * B;
}
return velocity;
```

```
}
```

C.11 Reversible Mass Action Reaction

```
long double Reaction_RevMassAction(long double kfun[], long double A, long double B,
long double P, long double Q, int numSvar, int numPvar)
{
int i;
long double k_scaled[2];
long double velocityF;
long double velocityR;
long double velocity;
for (i=0; i<2; i++){
if (kfun[i] <= .25){
k_scaled[i] = kfun[i] / .25 * 1e-2;
} else if (kfun[i] <= .50){
k_scaled[i] = (kfun[i] - .25) / .25 * 1e-1;
} else if (kfun[i] <= .75){
k_scaled[i] = (kfun[i] - .50) / .25;
} else {
k_scaled[i] = (kfun[i] - .75) / .25 * 1e1;
}
}
if (numSvar == 1){
velocityF = k_scaled[0] * A;
} else if (numSvar == 2) {
velocityF = k_scaled[0] * A * B;
}
if (numPvar == 1){
velocityR = k_scaled[1] * P;
```

```
} else if (numPvar == 2) {
velocityR = k_scaled[1] * P * Q;
}
velocity = velocityF - velocityR;
return velocity;
}
```

C.12 Michaelis Menten Reaction

```
long double Reaction_MichaelisM(long double k[], long double S, long double Et)
{
long double k1;
long double kn1;
long double k2;
long double KM;
long double Vmax;
long double velocity;
if(k[0] <= .33333){
     k1 = k[0]/.33333 * 1e6;
} else if( k[0] <= .66666) {
k1 = (k[0]-.33333)/.33333 * 1e7;
} else if(k[0] <= .99999) {
k1 = (k[0] - .66666)/.33333 * 1e8;
} else {
k1 = 1e8;
}
k1 = k1 * 3600 * 1e-12;
if(k[1] <= .25){
     kn1 = k[1]/.25 * 1e1;
} else if(k[1] <= .50) {
```

```
kn1 = (k[1]-.25)/.25 * 1e2;
} else if(k[1] <= .75) {
     kn1 = (k[1] - .50)/.25 * 1e3;
} else if(k[1] > .75) {
     kn1 = (k[1] - .75)/.25 * 1e4;
}
kn1 = kn1 * 3600;
if(k[2] <= .20){
     k2 = k[2]/.20 * 1e1;
} else if(k[2] <= .40){
     k2 = (k[2]-.20)/.20 * 1e2;
} else if(k[2] <= .60){
     k2 = (k[2] - .40)/.20 * 1e3;
} else if(k[2] <= .80){
k2 = (k[2] - .60)/.20 * 1e4;
} else if(k[2] > .80){
     k2 = (k[2] - .80)/.20 * 1e5;
}
k2 = k2 * 3600;
KM = (kn1 + k2) / k1;
Vmax = k2 * Et;
velocity = (Vmax * S) / (S + KM);
return velocity;

}
```

C.13 Reversible Michaelis Menten Reaction

```
long double Reaction_RevMichaelisM(long double k[], long double S,
long double P, long double Et)
```

```
{
long double k1;
long double kn1;
long double k2;
long double kn2;
long double KMS;
long double KMP;
long double KS;
long double KP;
long double velocity;
if(k[0] <= .33333){
     k1 = k[0]/.33333 * 1e6;
} else if( k[0] <= .66666) {
k1 = (k[0]-.33333)/.33333 * 1e7;
} else if(k[0] <= .99999) {
k1 = (k[0] - .66666)/.33333 * 1e8;
} else {
k1 = 1e8;
}
k1 = k1 * 3600 * 1e-12;
if(k[1] <= .25){
     kn1 = k[1]/.25 * 1e1;
} else if(k[1] <= .50) {
kn1 = (k[1]-.25)/.25 * 1e2;
} else if(k[1] <= .75) {
     kn1 = (k[1] - .50)/.25 * 1e3;
} else if(k[1] > .75) {
     kn1 = (k[1] - .75)/.25 * 1e4;
}
```

```
kn1 = kn1 * 3600;
if(k[2] <= .20){
     k2 = k[2]/.20 * 1e1;
} else if(k[2] <= .40){
     k2 = (k[2]-.20)/.20 * 1e2;
} else if(k[2] <= .60){
     k2 = (k[2] - .40)/.20 * 1e3;
} else if(k[2] <= .80){
k2 = (k[2] - .60)/.20 * 1e4;
} else if(k[2] > .80){
     k2 = (k[2] - .80)/.20 * 1e5;
}
k2 = k2 * 3600;
if(k[3] <= .20){
     kn2 = k[3]/.20 * 1e1;
} else if(k[3] <= .40){
     kn2 = (k[3]-.20)/.20 * 1e2;
} else if(k[3] <= .60){
     kn2 = (k[3] - .40)/.20 * 1e3;
} else if(k[3] <= .80){
kn2 = (k[3] - .60)/.20 * 1e4;
} else if(k[3] > .80){
     kn2 = (k[3] - .80)/.20 * 1e5;
}
kn2 = kn2 * 3600 * 1e-12;
KMS = (kn1 + k2) / k1;
KS = (k1 * k2) / (kn1 + k2);
KMP = (kn1 + k2) / kn2;
KP = (kn1 * kn2) / (kn1 + k2);
```

```
velocity = ((KS * Et * S) - (KP * Et * P))/(1.0 + S/KMS + P/KMP);
return velocity;

}
```

C.14 Uni Uni Reaction

```
long double Reaction_UniUni(long double kfun[], long double S,
long double P, long double Et)
{
int i;
long double k_unscaled[6];
long double num;
long double den1;
long double den2;
long double den3;
long double velocity;
for(i=0; i<6; i++)
{
     if(kfun[i] < .11)
     {
         k_unscaled[i]=kfun[i]*1e1;
         }
         else if (kfun[i]<.21)
         {
         k_unscaled[i]=(kfun[i]-.1)*1e2;
         }
         else if (kfun[i]<.31)
         {
         k_unscaled[i]=(kfun[i]-.2)*1e3;
```

```
}
else if (kfun[i]<.41)
{
k_unscaled[i]=(kfun[i]-.3)*1e4;
}
else if (kfun[i]<.51)
{
k_unscaled[i]=(kfun[i]-.4)*1e5;
}
else if (kfun[i]<.61)
{
k_unscaled[i]=(kfun[i]-.5)*1e6;
}
else if (kfun[i]<.71)
{
k_unscaled[i]=(kfun[i]-.6)*1e7;
}
else if (kfun[i]<.81)
{
k_unscaled[i]=(kfun[i]-.7)*1e8;
}
else if (kfun[i]<.91)
{
k_unscaled[i]=(kfun[i]-.8)*1e9;
}
else if (kfun[i]<= 1)
{
k_unscaled[i]=(kfun[i]-.9)*1e10;
}
```

```
    }
    k_unscaled[0] = k_unscaled[0] * 3600 * 1e-12;
    k_unscaled[1] = k_unscaled[1] * 3600;
    k_unscaled[2] = k_unscaled[2] * 3600;
    k_unscaled[3] = k_unscaled[3] * 3600;
    k_unscaled[4] = k_unscaled[4] * 3600;
    k_unscaled[5] = k_unscaled[5] * 3600 * 1e-12;
    num = (k_unscaled[0] * k_unscaled[2] * k_unscaled[4] * S -
    k_unscaled[1] * k_unscaled[3] * k_unscaled[5] * P) * Et;
    den1 = k_unscaled[1] * k_unscaled[4] +
    k_unscaled[1] * k_unscaled[3] + k_unscaled[2] * k_unscaled[4];
    den2 = S * k_unscaled[0] * (k_unscaled[2] + k_unscaled[3] + k_unscaled[4]);
    den3 = P * k_unscaled[5] * (k_unscaled[1] + k_unscaled[2] + k_unscaled[3]);
    velocity = num / (den1 + den2 + den3);
    return velocity;
}
```

C.15 Generalized Mass Action Reaction

```
long double Reaction_GMA(long double k[], long double A, long double B, int numSvar)
{
int i;
long double kf;
long double g[2];
long double velocity;
if(k[0] <= .33333){
     kf = k[0]/.33333 * 1e1;
} else if( k[0] <= .66666) {
kf = (k[0]-.33333)/.33333 * 1e2;
} else if(k[0] <= .99999) {
```

```
kf = (k[0] - .66666)/.33333 * 1e3;
} else {
kf = 1e3;
}
for(i=0; i<2; i++){
if(k[i + 1] <= .33333){
     g[i] = k[i+1]/.33333 * 1.0;
} else if( k[i +1] <= .66666) {
g[i] = (k[i +1]-.33333)/.33333 * 2.0;
} else if(k[i +1] <= .99999) {
g[i] = (k[i +1] - .66666)/.33333 * 3.0;
} else {
g[i] = 3.0;
}
}
if (numSvar == 1){
if (A < 1.0e-8){
velocity = 0.0;
} else {
velocity = kf * pow(A, g[0]);
}
} else if (numSvar == 2) {
if ((A < 1.0e-8) || (B < 1.0e-8)){
velocity = 0.0;
} else {
velocity = kf * pow(A, g[0]) * pow(B, g[1]);
}
}
return velocity;
```

```
}
```

C.16 Reversible Generalized Mass Action Reaction

```
long double Reaction_RevGMA(long double k[], long double A, long double B,
long double P, long double Q, int numSvar, int numPvar)
{
int i;
long double kf;
long double kr;
long double g[4];
long double velocity;
long double velocityF;
long double velocityR;
if(k[0] <= .33333){
     kf = k[0]/.33333 * 1e1;
} else if( k[0] <= .66666) {
kf = (k[0]-.33333)/.33333 * 1e2;
} else if(k[0] <= .99999) {
kf = (k[0] - .66666)/.33333 * 1e3;
} else {
kf = 1e3;
}
if(k[1] <= .33333){
     kr = k[1]/.33333 * 1e1;
} else if( k[1] <= .66666) {
kr = (k[1]-.33333)/.33333 * 1e2;
} else if(k[1] <= .99999) {
kr = (k[1] - .66666)/.33333 * 1e3;
} else {
```

```
kr = 1e3;
}
for(i=0; i<(numSvar + numPvar); i++){
if(k[i + 1] <= .33333){
     g[i] = k[i+1]/.33333 * 1.0;
} else if( k[i +1] <= .66666) {
g[i] = (k[i +1]-.33333)/.33333 * 2.0;
} else if(k[i +1] <= .99999) {
g[i] = (k[i +1] - .66666)/.33333 * 3.0;
} else {
g[i] = 3.0;
}
}
if (numSvar == 1){
if (A < 1.0e-8){
velocityF = 0.0;
} else {
velocityF = kf * pow(A, g[0]);
}
} else if (numSvar == 2) {
if ((A < 1.0e-8) || (B < 1.0e-8)){
velocityF = 0.0;
} else {
velocityF = kf * pow(A, g[0]) * pow(B, g[1]);
}
}
if (numPvar == 1){
if (P < 1.0e-8){
velocityR = 0.0;
```

```
} else {
velocityR = kr * pow(P, g[numSvar]);
}
} else if (numPvar == 2) {
if ((P < 1.0e-8) || (Q < 1.0e-8)){
velocityR = 0.0;
} else {
velocityR = kr * pow(P, g[numSvar]) * pow(Q, g[numSvar+1]);
}
}
velocity = velocityF - velocityR;
return velocity;
}
```

C.17 Mass Action Substrate Derivative

```
long double kinetic_MassActionSder(long double kfun[], long double A,
long double B, int numSvar)
{
long double k_scaled;
long double deriv;
if (kfun[0] <= .25){
k_scaled = kfun[0] / .25 * 1e-2;
} else if (kfun[0] <= .50){
k_scaled = (kfun[0] - .25) / .25 * 1e-1;
} else if (kfun[0] <= .75){
k_scaled = (kfun[0] - .50) / .25;
} else {
k_scaled = (kfun[0] - .75) / .25 * 1e1;
}
```

```
if (numSvar == 1){
deriv = k_scaled;
} else if (numSvar == 2) {
deriv = k_scaled * B;
}
return deriv;
}
```

C.18 Reversible Mass Action Substrate Derivative

```
long double kinetic_RevMassActionSder(long double kfun[], long double A,
long double B, long double P, long double Q, int numSvar, int numPvar)
{
int i;
long double k_scaled[2];
long double deriv = 0.0;
for (i=0; i<2; i++){
if (kfun[i] <= .25){
k_scaled[i] = kfun[i] / .25 * 1e-2;
} else if (kfun[i] <= .50){
k_scaled[i] = (kfun[i] - .25) / .25 * 1e-1;
} else if (kfun[i] <= .75){
k_scaled[i] = (kfun[i] - .50) / .25;
} else {
k_scaled[i] = (kfun[i] - .75) / .25 * 1e1;
}
}
if (numSvar == 1){
deriv = k_scaled[0];
```

```
} else if (numSvar == 2) {
deriv = k_scaled[0] * B;
}
return deriv;
}
```

C.19 Reversible Mass Action Product Derivative

```
long double kinetic_RevMassActionPder(long double kfun[], long double A,
long double B, long double P, long double Q, int numSvar, int numPvar)
{
int i;
long double k_scaled[2];
long double deriv;
for (i=0; i<2; i++){
if (kfun[i] <= .25){
k_scaled[i] = kfun[i] / .25 * 1e-2;
} else if (kfun[i] <= .50){
k_scaled[i] = (kfun[i] - .25) / .25 * 1e-1;
} else if (kfun[i] <= .75){
k_scaled[i] = (kfun[i] - .50) / .25;
} else {
k_scaled[i] = (kfun[i] - .75) / .25 * 1e1;
}
}
if (numPvar == 1){
deriv = k_scaled[1];
} else if (numPvar == 2) {
deriv = k_scaled[1] * Q;
}
```

```
return deriv;
}
```

C.20 Michaelis Menten Substrate Derivative

```
long double kinetic_MichaelisSder(long double k[], long double S, long double Et)
{
long double k1;
long double kn1;
long double k2;
long double KM;
long double Vmax;
long double num;
long double denom;
long double numderiv;
long double denomderiv;
long double deriv;
if(k[0] <= .33333){
     k1 = k[0]/.33333 * 1e6;
} else if( k[0] <= .66666) {
k1 = (k[0]-.33333)/.33333 * 1e7;
} else if(k[0] <= .99999) {
k1 = (k[0] - .66666)/.33333 * 1e8;
} else {
k1 = 1e8;
}
k1 = k1 * 3600 * 1e-12;
if(k[1] <= .25){
     kn1 = k[1]/.25 * 1e1;
} else if(k[1] <= .50) {
```

```
kn1 = (k[1]-.25)/.25 * 1e2;
} else if(k[1] <= .75) {
     kn1 = (k[1] - .50)/.25 * 1e3;
} else if(k[1] > .75) {
     kn1 = (k[1] - .75)/.25 * 1e4;
}
kn1 = kn1 * 3600;
if(k[2] <= .20){
     k2 = k[2]/.20 * 1e1;
} else if(k[2] <= .40){
     k2 = (k[2]-.20)/.20 * 1e2;
} else if(k[2] <= .60){
     k2 = (k[2] - .40)/.20 * 1e3;
} else if(k[2] <= .80){
k2 = (k[2] - .60)/.20 * 1e4;
} else if(k[2] > .80){
     k2 = (k[2] - .80)/.20 * 1e5;
}
k2 = k2 * 3600;
KM = (kn1 + k2) / k1;
Vmax = k2 * Et;
num = (Vmax * S);
denom = (S + KM);
numderiv = Vmax;
denomderiv = 0.0;
deriv = (numderiv * denom - num * denomderiv) / (denom * denom);
return deriv;
}
```

C.21 Reversible Michaelis Menten Substrate Derivative

```
long double kinetic_RevMichaelisSder(long double k[], long double S,
long double P, long double Et)
{
long double k1;
long double kn1;
long double k2;
long double kn2;
long double KMS;
long double KMP;
long double KS;
long double KP;
long double num;
long double denom;
long double numderiv;
long double denomderiv;
long double deriv;
if(k[0] <= .33333){
     k1 = k[0]/.33333 * 1e6;
} else if( k[0] <= .66666) {
k1 = (k[0]-.33333)/.33333 * 1e7;
} else if(k[0] <= .99999) {
k1 = (k[0] - .66666)/.33333 * 1e8;
} else {
k1 = 1e8;
}
k1 = k1 * 3600 * 1e-12;
if(k[1] <= .25){
     kn1 = k[1]/.25 * 1e1;
```

```
} else if(k[1] <= .50) {
kn1 = (k[1]-.25)/.25 * 1e2;
} else if(k[1] <= .75) {
     kn1 = (k[1] - .50)/.25 * 1e3;
} else if(k[1] > .75) {
     kn1 = (k[1] - .75)/.25 * 1e4;
}
kn1 = kn1 * 3600;
if(k[2] <= .20){
     k2 = k[2]/.20 * 1e1;
} else if(k[2] <= .40){
     k2 = (k[2]-.20)/.20 * 1e2;
} else if(k[2] <= .60){
     k2 = (k[2] - .40)/.20 * 1e3;
} else if(k[2] <= .80){
k2 = (k[2] - .60)/.20 * 1e4;
} else if(k[2] > .80){
     k2 = (k[2] - .80)/.20 * 1e5;
}
k2 = k2 * 3600;
if(k[3] <= .20){
     kn2 = k[3]/.20 * 1e1;
} else if(k[3] <= .40){
     kn2 = (k[3]-.20)/.20 * 1e2;
} else if(k[3] <= .60){
     kn2 = (k[3] - .40)/.20 * 1e3;
} else if(k[3] <= .80){
kn2 = (k[3] - .60)/.20 * 1e4;
} else if(k[3] > .80){
```

```
     kn2 = (k[3] - .80)/.20 * 1e5;
}
kn2 = kn2 * 3600 * 1e-12;
KMS = (kn1 + k2) / k1;
KS = (k1 * k2) / (kn1 + k2);
KMP = (kn1 + k2) / kn2;
KP = (kn1 * kn2) / (kn1 + k2);
num = ((KS * Et * S) - (KP * Et * P));
denom = (1.0 + S/KMS + P/KMP);
numderiv = KS * Et;
denomderiv = 1.0/KMS;
deriv = (numderiv * denom - num * denomderiv) / (denom * denom);
return deriv;
}
```

C.22 Reversible Michaelis Menten Product Derivative

```
long double kinetic_RevMichaelisPder(long double k[], long double S,
long double P, long double Et)
{
long double k1;
long double kn1;
long double k2;
long double kn2;
long double KMS;
long double KMP;
long double KS;
long double KP;
long double num;
long double denom;
```

```
long double numderiv;
long double denomderiv;
long double deriv;
if(k[0] <= .33333){
     k1 = k[0]/.33333 * 1e6;
} else if( k[0] <= .66666) {
k1 = (k[0]-.33333)/.33333 * 1e7;
} else if(k[0] <= .99999) {
k1 = (k[0] - .66666)/.33333 * 1e8;
} else {
k1 = 1e8;
}
k1 = k1 * 3600 * 1e-12;
if(k[1] <= .25){
     kn1 = k[1]/.25 * 1e1;
} else if(k[1] <= .50) {
kn1 = (k[1]-.25)/.25 * 1e2;
} else if(k[1] <= .75) {
     kn1 = (k[1] - .50)/.25 * 1e3;
} else if(k[1] > .75) {
     kn1 = (k[1] - .75)/.25 * 1e4;
}
kn1 = kn1 * 3600;
if(k[2] <= .20){
     k2 = k[2]/.20 * 1e1;
} else if(k[2] <= .40){
     k2 = (k[2]-.20)/.20 * 1e2;
} else if(k[2] <= .60){
     k2 = (k[2] - .40)/.20 * 1e3;
```

```
} else if(k[2] <= .80){
k2 = (k[2] - .60)/.20 * 1e4;
} else if(k[2] > .80){
     k2 = (k[2] - .80)/.20 * 1e5;
}
k2 = k2 * 3600;
if(k[3] <= .20){
     kn2 = k[3]/.20 * 1e1;
} else if(k[3] <= .40){
     kn2 = (k[3]-.20)/.20 * 1e2;
} else if(k[3] <= .60){
     kn2 = (k[3] - .40)/.20 * 1e3;
} else if(k[3] <= .80){
kn2 = (k[3] - .60)/.20 * 1e4;
} else if(k[3] > .80){
     kn2 = (k[3] - .80)/.20 * 1e5;
}
kn2 = kn2 * 3600 * 1e-12;
KMS = (kn1 + k2) / k1;
KS = (k1 * k2) / (kn1 + k2);
KMP = (kn1 + k2) / kn2;
KP = (kn1 * kn2) / (kn1 + k2);
num = ((KS * Et * S) - (KP * Et * P));
denom = (1.0 + S/KMS + P/KMP);
numderiv = - (KP * Et);
denomderiv = 1.0/KMP;
deriv = (numderiv * denom - num * denomderiv) / (denom * denom);
return deriv;
}
```

C.23 Uni Uni Substrate Derivative

```
long double kinetic_uniunisder(long double k[],long double S,
long double P, long double Et)
{
int i;
long double k_unscaled[6];
long double u;
long double u_prime;
long double v;
long double v_prime;
long double num;
long double den;
for(i=0; i<6; i++)
{
     if(k[i] < .11)
     {
         k_unscaled[i]=k[i]*1e1;
         }
         else if (k[i]<.21)
         {
         k_unscaled[i]=(k[i]-.1)*1e2;
         }
         else if (k[i]<.31)
         {
         k_unscaled[i]=(k[i]-.2)*1e3;
         }
         else if (k[i]<.41)
         {
         k_unscaled[i]=(k[i]-.3)*1e4;
```

```
    }
    else if (k[i]<.51)
    {
    k_unscaled[i]=(k[i]-.4)*1e5;
    }
    else if (k[i]<.61)
    {
    k_unscaled[i]=(k[i]-.5)*1e6;
    }
    else if (k[i]<.71)
    {
    k_unscaled[i]=(k[i]-.6)*1e7;
    }
    else if (k[i]<.81)
    {
    k_unscaled[i]=(k[i]-.7)*1e8;
    }
    else if (k[i]<.91)
    {
    k_unscaled[i]=(k[i]-.8)*1e9;
    }
    else if (k[i]<= 1)
    {
    k_unscaled[i]=(k[i]-.9)*1e10;
    }
}
k_unscaled[0] = k_unscaled[0] * 3600 * 1e-12;
k_unscaled[1] = k_unscaled[1] * 3600;
k_unscaled[2] = k_unscaled[2] * 3600;
```

```
    k_unscaled[3] = k_unscaled[3] * 3600;
    k_unscaled[4] = k_unscaled[4] * 3600;
    k_unscaled[5] = k_unscaled[5] * 3600 * 1e-12;
    u_prime = k_unscaled[0] * k_unscaled[2] * k_unscaled[4] * Et;
    v = k_unscaled[1]*k_unscaled[4] + k_unscaled[1]*k_unscaled[3]
    + k_unscaled[2]*k_unscaled[4] + S*k_unscaled[0]*(k_unscaled[5]
    + k_unscaled[3] + k_unscaled[2]) + P*k_unscaled[5]
    *(k_unscaled[1] + k_unscaled[2] + k_unscaled[3]);
    v_prime = k_unscaled[0]*(k_unscaled[2] + k_unscaled[4] + k_unscaled[5]);
    u = k_unscaled[0]*k_unscaled[2]*k_unscaled[4]*S*Et
    - k_unscaled[1]*k_unscaled[3]*k_unscaled[5]*P*Et;
    num = u_prime * v - v_prime * u;
    den = pow(v, 2.0);
return num/den;
}
```

C.24 Uni Uni Product Derivative

```
long double kinetic_uniunipder(long double k[],long double S,
long double P,long double Et)
{
int i;
long double k_unscaled[6];
long double u;
long double u_prime;
long double v;
long double v_prime;
long double num;
long double den;
for(i=0; i<6; i++)
```

```
{
     if(k[i] < .11)
     {
         k_unscaled[i]=k[i]*1e1;
         }
         else if (k[i]<.21)
         {
         k_unscaled[i]=(k[i]-.1)*1e2;
         }
         else if (k[i]<.31)
         {
         k_unscaled[i]=(k[i]-.2)*1e3;
         }
         else if (k[i]<.41)
         {
         k_unscaled[i]=(k[i]-.3)*1e4;
         }
         else if (k[i]<.51)
         {
         k_unscaled[i]=(k[i]-.4)*1e5;
         }
         else if (k[i]<.61)
         {
         k_unscaled[i]=(k[i]-.5)*1e6;
         }
         else if (k[i]<.71)
         {
         k_unscaled[i]=(k[i]-.6)*1e7;
         }
```

```
      else if (k[i]<.81)
      {
      k_unscaled[i]=(k[i]-.7)*1e8;
      }
      else if (k[i]<.91)
      {
      k_unscaled[i]=(k[i]-.8)*1e9;
      }
      else if (k[i]<= 1)
      {
      k_unscaled[i]=(k[i]-.9)*1e10;
      }
}
k_unscaled[0] = k_unscaled[0] * 3600 * 1e-12;
k_unscaled[1] = k_unscaled[1] * 3600;
k_unscaled[2] = k_unscaled[2] * 3600;
k_unscaled[3] = k_unscaled[3] * 3600;
k_unscaled[4] = k_unscaled[4] * 3600;
k_unscaled[5] = k_unscaled[5] * 3600 * 1e-12;
u_prime = -k_unscaled[1] * k_unscaled[3] * k_unscaled[5] * Et;
v = k_unscaled[1]*k_unscaled[4] + k_unscaled[1]*k_unscaled[3]
+ k_unscaled[2]*k_unscaled[4] + S*k_unscaled[0]*(k_unscaled[5]
+ k_unscaled[3] + k_unscaled[2]) + P*k_unscaled[5]*(k_unscaled[1]
+ k_unscaled[2] + k_unscaled[3]);
v_prime = k_unscaled[5]*(k_unscaled[1] + k_unscaled[2] + k_unscaled[3]);
u = k_unscaled[0]*k_unscaled[2]*k_unscaled[4]*S*Et
- k_unscaled[1]*k_unscaled[3]*k_unscaled[5]*P*Et;
num = u_prime * v - v_prime * u;
den = pow(v, 2.0);
```

```
    return num/den;
}
```

C.25 Generalized Mass Action Substrate Derivative

```
long double kinetic_GMASder(long double k[], long double A,
long double B, int numSvar)
{
int i;
long double kf;
long double g[2];
long double deriv;
if(k[0] <= .33333){
     kf = k[0]/.33333 * 1e1;
} else if( k[0] <= .66666) {
kf = (k[0]-.33333)/.33333 * 1e2;
} else if(k[0] <= .99999) {
kf = (k[0] - .66666)/.33333 * 1e3;
} else {
kf = 1e3;
}
for(i=0; i<2; i++){
if(k[i + 1] <= .33333){
     g[i] = k[i+1]/.33333 * 1.0;
} else if( k[i +1] <= .66666) {
g[i] = (k[i +1]-.33333)/.33333 * 2.0;
} else if(k[i +1] <= .99999) {
g[i] = (k[i +1] - .66666)/.33333 * 3.0;
} else {
g[i] = 3.0;
```

```
}
}
if (numSvar == 1){
if (A < 1.0e-8){
deriv = 0.0;
} else {
deriv = kf * g[0] * pow(A, g[0] - 1.0);
}
} else if (numSvar == 2) {
if ((A < 1.0e-8) || (B < 1.0e-8)){
deriv = 0.0;
} else {
deriv = kf * g[0] * pow(A, g[0] - 1.0) * pow(B, g[1]);
}
}
return deriv;
}
```

C.26 Reversible Generalized Mass Action Substrate Derivative

```
long double kinetic_RevGMASder(long double k[], long double A, long double B,
long double P, long double Q, int numSvar, int numPvar)
{
int i;
long double kf;
long double kr;
long double g[4];
long double deriv = 0.0;
if(k[0] <= .33333){
```

```
     kf = k[0]/.33333 * 1e1;
} else if( k[0] <= .66666) {
kf = (k[0]-.33333)/.33333 * 1e2;
} else if(k[0] <= .99999) {
kf = (k[0] - .66666)/.33333 * 1e3;
} else {
kf = 1e3;
}
if(k[1] <= .33333){
     kr = k[1]/.33333 * 1e1;
} else if( k[1] <= .66666) {
kr = (k[1]-.33333)/.33333 * 1e2;
} else if(k[1] <= .99999) {
kr = (k[1] - .66666)/.33333 * 1e3;
} else {
kr = 1e3;
}
for(i=0; i<(numSvar + numPvar); i++){
if(k[i + 1] <= .33333){
     g[i] = k[i+1]/.33333 * 1.0;
} else if( k[i +1] <= .66666) {
g[i] = (k[i +1]-.33333)/.33333 * 2.0;
} else if(k[i +1] <= .99999) {
g[i] = (k[i +1] - .66666)/.33333 * 3.0;
} else {
g[i] = 3.0;
}
}
if (numSvar == 1){
```

```
if (A < 1.0e-8){
deriv = 0.0;
} else {
deriv = kf * g[0] * pow(A, g[0] - 1.0);
}
} else if (numSvar == 2) {
if ((A < 1.0e-8) || (B < 1.0e-8)){
deriv = 0.0;
} else {
deriv = kf * g[0] * pow(A, g[0] - 1.0) * pow(B, g[1]);
}
}
return deriv;
}
```

C.27 Reversible Generalized Mass Action Product Derivative

```
long double kinetic_RevGMAPder(long double k[], long double A, long double B,
long double P, long double Q, int numSvar, int numPvar)
{
int i;
long double kf;
long double kr;
long double g[4];
long double deriv;
if(k[0] <= .33333){
     kf = k[0]/.33333 * 1e1;
} else if( k[0] <= .66666) {
kf = (k[0]-.33333)/.33333 * 1e2;
```

```
} else if(k[0] <= .99999) {
kf = (k[0] - .66666)/.33333 * 1e3;
} else {
kf = 1e3;
}
if(k[1] <= .33333){
     kr = k[1]/.33333 * 1e1;
} else if( k[1] <= .66666) {
kr = (k[1]-.33333)/.33333 * 1e2;
} else if(k[1] <= .99999) {
kr = (k[1] - .66666)/.33333 * 1e3;
} else {
kr = 1e3;
}
for(i=0; i<(numSvar + numPvar); i++){
if(k[i + 1] <= .33333){
     g[i] = k[i+1]/.33333 * 1.0;
} else if( k[i +1] <= .66666) {
g[i] = (k[i +1]-.33333)/.33333 * 2.0;
} else if(k[i +1] <= .99999) {
g[i] = (k[i +1] - .66666)/.33333 * 3.0;
} else {
g[i] = 3.0;
}
}
if (numPvar == 1){
if (P < 1.0e-8){
deriv = 0.0;
} else {
```

```
deriv = kr * g[numSvar] * pow(P, g[numSvar] - 1.0);
}
} else if (numPvar == 2) {
if ((P < 1.0e-8) || (Q < 1.0e-8)){
deriv = 0.0;
} else {
deriv = kr * g[numSvar] * pow(P, g[numSvar] - 1.0) * pow(Q, g[numSvar+1]);
}
}
return deriv;
}
```

C.28 Random Number Generator

```
long double ran2(long *idum)
{
int j;
long k;
static long idum2=123456789;
static long iy=0;
static long iv[32];
long double temp;
long double am = (1.0/2147483563);
long imm1 = (2147483563-1);
long NDIV = (1+imm1/32);
long double eps = 1.2e-7;
long double RNMX  = (1.0-eps);
static long IA1 = 40014;
static long IA2 = 40692;
static long IQ1 = 53668;
```

```
static long IQ2 = 52774;
static long IR1 = 12211;
static long IR2 = 3791;
if (*idum <= 0) {
if (-(*idum) < 1) *idum=1;
else *idum = -(*idum);
idum2=(*idum);
for (j=32+7;j>=0;j--) {
k=(*idum)/IQ1;
*idum=IA1*(*idum-k*IQ1)-k*IR1;
if (*idum < 0) *idum += 2147483563;
if (j < 32) iv[j] = *idum;
}
iy=iv[0];
}
k=(*idum)/IQ1;
*idum=IA1*(*idum-k*IQ1)-k*IR1;
if (*idum < 0) *idum += 2147483563;
k=idum2/IQ2;
idum2=IA2*(idum2-k*IQ2)-k*IR2;
if (idum2 < 0) idum2 += 2147483399;
j=iy/NDIV;
iy=iv[j]-idum2;
iv[j] = *idum;
if (iy < 1) iy += imm1;
if ((temp=am*iy) > RNMX) return RNMX;
else return temp;
}
```

APPENDIX D

HIGH PERFORMANCE COMPUTING GENETIC ALGORITHM C SOURCE

This appendix contains the additional C/MPI source necessary to transform the Monte Carlo Source into a fitting algorithm that uses a Genetic Algorithm.

D.1 Genetic Algorithm Driver

```
int GeneticAlgorithmDriver(int iter, long double k[], long double X[], int F[],
int R[], long double Et[], int numVar, int neq, int ncn, int timepts,
long seed, char save_root[], int id, int Popsize, long double MRate,
int EliteCount, int p)
{
struct timeval tp1, tp2;
long double *OldPop;
long double *NewPop;
long double *Scores;
long double *Expect;
long double *Elite;
long double *EliteScores;
int *RankedIndex;
int nParents = Popsize * 2 - EliteCount;
int *Parents;
int RowsX = timepts;
int ColsX = neq + 1;
long double timespan[timepts];
long double Y_Zero[neq];
```

```
int count = 0;
int i;
int j;
int eliteindex =0;
long double Cost;
long double BestCost = 1.0e7;
MPI_Status status;
int timeRecord = 0;
int loopStop = 0;
int loopCount = 0;
OldPop = malloc(numVar * Popsize * sizeof(long double));
NewPop = malloc(numVar * Popsize * sizeof(long double));
Scores = malloc(Popsize * sizeof(long double));
Expect = malloc(Popsize * sizeof(long double));
Elite = malloc(numVar * EliteCount * sizeof(long double));
EliteScores = malloc(EliteCount * sizeof(long double));
RankedIndex = malloc(Popsize * sizeof(int));
Parents = malloc(nParents * sizeof(int));
gettimeofday(&tp1, NULL);
if (id == 0) {
int process_num;
int finished = p - 1;
char errorFile[37] = "Error_";
char paramFile[37] = "Param_";
char timeFile[37] = "Time_";
FILE *pWrite1;
FILE *pWrite2;
FILE *pWrite3;
int proccessFinished[p];
```

```
for (i=0; i<p; i++){
proccessFinished[i] = 0;
}
sprintf(idname,"%d", id);
strncat(errorFile, save_root, 36 - strlen(errorFile));
strncat(paramFile, save_root, 36 - strlen(paramFile));
strncat(timeFile, save_root, 36 - strlen(timeFile));
strncat(errorFile, idname, 36 - strlen(errorFile));
strncat(paramFile, idname, 36 - strlen(paramFile));
while (loopCount < p-1){
MPI_Recv(&process_num, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
MPI_Recv(&Cost, 1, MPI_LONG_DOUBLE, process_num, 0, MPI_COMM_WORLD, &status);
MPI_Recv(k, numVar, MPI_LONG_DOUBLE, process_num, 0, MPI_COMM_WORLD, &status);
MPI_Recv(&i, 1, MPI_INT, process_num, 0, MPI_COMM_WORLD, &status);
MPI_Recv(&timeRecord, 1, MPI_INT, process_num, 0, MPI_COMM_WORLD, &status);
//printf("I'm recieved message on iter %d\n", i);
pWrite1 = fopen(errorFile, "a");
pWrite2 = fopen(paramFile, "a");
pWrite3 = fopen(timeFile, "a");
if (Cost < BestCost) {
fprintf(pWrite1, "%Le\n", Cost);
for(j=0; j<numVar; j++){
fprintf(pWrite2, "%Lf\t", k[j]);
}
fprintf(pWrite2, "\n");
fprintf(pWrite3, "%d\n", timeRecord);
BestCost = Cost;
}
fclose(pWrite1);
```

```
fclose(pWrite2);
fclose(pWrite3);
//printf("i is %d\n", i);
//printf("iter is %d\n", iter);
if ((i == iter)&&(proccessFinished[process_num] == 0)){
proccessFinished[process_num] = 1;
finished = finished - 1;
printf("Finished = %d\n", finished);
}
if (finished == 0) {
loopStop = 1;
loopCount = loopCount + 1;
printf("loopCount is %d\n", loopCount);
}
MPI_Send(&loopStop, 1, MPI_INT, process_num, 0, MPI_COMM_WORLD);
//printf("Finished is %d\n", finished);
}
gettimeofday(&tp2, NULL);
printf("%d\n", (int)(tp2.tv_sec - tp1.tv_sec));
} else {
seed = seed - id;
for(i=0; i<timepts; i++){
timespan[i] = X[i * ColsX];
}
for(i=0; i<neq; i++){
Y_Zero[i] = X[i+1];
}
while (loopStop == 0){
//Initial Population Creation and Scoring
```

```
for(i=0;i<Popsize;i++){
for(j=0; j<numVar; j++){
k[j] = ran2(&seed);
OldPop[i * numVar + j] = k[j];
}
Scores[i] = CostFunction(k, X, F, R, Et, neq, ncn, RowsX, ColsX, timespan, Y_Zero);
Cost = Scores[i];
RankedIndex[i] = i;
if ((Cost < BestCost) && (i != iter)){
//printf("I'm sending message on iter %d\n", i);
gettimeofday(&tp2, NULL);
timeRecord = (int)(tp2.tv_sec - tp1.tv_sec);
MPI_Send(&id, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&Cost, 1, MPI_LONG_DOUBLE, 0, 0, MPI_COMM_WORLD);
MPI_Send(k, numVar, MPI_LONG_DOUBLE, 0, 0, MPI_COMM_WORLD);
MPI_Send(&count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&timeRecord, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Recv(&loopStop, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
BestCost = Cost;
}
}
//Genetic Algorithm Loop
for (count=1; count<iter; count++){
//Call Scaling Function
FitScaling (Expect, Scores, RankedIndex, nParents, Popsize);
//Identifying Elite Members
for (i=0; i<EliteCount; i++){
for(j=0; j<numVar; j++){
Elite[i * numVar + j] = OldPop[RankedIndex[i] * numVar + j];
```

```
}
EliteScores[i] = Scores[i];
}
//Call Selection Function and Shuffle Parents
Selection (Parents, Expect, nParents, Popsize, seed);
//Call Recombination/Mutation Function and Evaluate Fitness
RecombMutFit (NewPop, OldPop, Scores, Parents, nParents, MRate, seed, numVar);
//Replacing Old Generation with Newly Created One and insert Elite Mememberss
eliteindex = 0;
for(i=0;i<Popsize;i++){
if (i < (Popsize - EliteCount)){
for(j=0; j<numVar; j++){
OldPop[i * numVar + j] = NewPop[i * numVar + j];
}
Scores[i] = CostFunction(k, X, F, R, Et, neq, ncn, RowsX, ColsX, timespan, Y_Zero);
Cost = Scores[i];
if ((Cost < BestCost) && (i != iter)){
//printf("I'm sending message on iter %d\n", i);
gettimeofday(&tp2, NULL);
timeRecord = (int)(tp2.tv_sec - tp1.tv_sec);
MPI_Send(&id, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&Cost, 1, MPI_LONG_DOUBLE, 0, 0, MPI_COMM_WORLD);
MPI_Send(k, numVar, MPI_LONG_DOUBLE, 0, 0, MPI_COMM_WORLD);
MPI_Send(&count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&timeRecord, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Recv(&loopStop, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
BestCost = Cost;
}
} else {
```

```
for(j=0; j<numVar; j++){
OldPop[i * numVar + j] = Elite[eliteindex * numVar + j];
}
Scores[i] = EliteScores[eliteindex];
Cost = Scores[i];
eliteindex = eliteindex + 1;
if ((Cost < BestCost) && (i != iter)){
//printf("I'm sending message on iter %d\n", i);
gettimeofday(&tp2, NULL);
timeRecord = (int)(tp2.tv_sec - tp1.tv_sec);
MPI_Send(&id, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&Cost, 1, MPI_LONG_DOUBLE, 0, 0, MPI_COMM_WORLD);
MPI_Send(k, numVar, MPI_LONG_DOUBLE, 0, 0, MPI_COMM_WORLD);
MPI_Send(&count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&timeRecord, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Recv(&loopStop, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
BestCost = Cost;
}
}
RankedIndex[i] = i;
}
//Migration?
}
loopCount = loopCount + 1;
printf("I've completed the GA on process %d %d times\n", id, loopCount);
gettimeofday(&tp2, NULL);
timeRecord = (int)(tp2.tv_sec - tp1.tv_sec);
MPI_Send(&id, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&Cost, 1, MPI_LONG_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

```
MPI_Send(k, numVar, MPI_LONG_DOUBLE, 0, 0, MPI_COMM_WORLD);
MPI_Send(&count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&timeRecord, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Recv(&loopStop, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
}
}
//Memory Deallocation
free(OldPop);
free(NewPop);
free(Scores);
free(Expect);
free(Elite);
free(EliteScores);
free(RankedIndex);
free(Parents);
OldPop = NULL;
NewPop = NULL;
Scores = NULL;
Expect = NULL;
Elite = NULL;
EliteScores = NULL;
RankedIndex = NULL;
Parents = NULL;
return 0;
}
```

D.2 Fitness Scaling Function

```
int FitScaling (long double Expect[], long double Scores[], int index[],
int nParents, int Popsize)
```

```
{
int i;
long double iReal;
long double sumExpect = 0.0;
long double cumsum = 0.0;

sort2(Popsize, Scores, index);

for(i=0; i<Popsize; i++){
iReal = i+1;
Expect[i] = pow(iReal, 0.5) / iReal;
sumExpect = sumExpect + Expect[i];
}

for(i=0; i<Popsize; i++){
Expect[i] = (nParents * Expect[i]) / sumExpect;
cumsum += Expect[i];
Expect[i] = cumsum / nParents;
}

return 0;
}
```

D.3 Selection Function

```
int Selection (int Parents[], long double wheel[], int nParents,
int Popsize, long seed)
{
int i;
int j;
```

```
int lowest = 0;
long double shuffle[nParents];
long double stepSize = nParents;
long double position;

stepSize = 1.0/stepSize;
position = ran2(&seed) * stepSize;

for (i=0; i<nParents; i++){
for (j=lowest; j<Popsize; j++){
if (position < wheel[j]) {
Parents[i] = j;
lowest = j;
break;
}
}
position = position + stepSize;
shuffle[i] = ran2(&seed);
}

//Shuffle Parents
sort2(nParents, shuffle, Parents);

return 0;
}
```

D.4 Combined Recombination and Mutation Function

```
int RecombMutFit (long double NewPop[], long double OldPop[], long double Scores[],
int Parents[], int nParents, long double MRate, long seed, int numVar)
```

```
{
int i;
int j;
int r1;
int r2;
int index = 0;
int nKids = nParents / 2;

for (i=0; i<nKids; i++){
//Get Parents
r1 = Parents[index];
index = index + 1;
r2 = Parents[index];
index = index + 1;

//Randomly select half of the genes from each parent
for (j=0; j<numVar; j++){
if (ran2(&seed) <= MRate ){
NewPop[i * numVar + j] = ran2(&seed);
} else {
if (ran2(&seed) > 0.5){
NewPop[i * numVar + j] = OldPop[r1 * numVar + j];
} else {
NewPop[i * numVar + j] = OldPop[r2 * numVar + j];
}
}
}
}
return 0;
```

```
}
```

REFERENCES

[1] Alberts, B., *Molecular biology of the cell*. New York: Garland Science, 4th ed., 2002.

[2] Alon, U., Surette, M. G., Barkai, N., and Leibler, S., "Robustness in bacterial chemotaxis," *Nature*, vol. 397, no. 6715, pp. 168–171, 1999.

[3] Alvarez-Vasquez, F., Sims, K. J., Cowart, L. A., Okamoto, Y., Voit, E. O., and Hannun, Y. A., "Simulation and validation of modelled sphingolipid metabolism in saccharomyces cerevisiae," *Nature*, vol. 433, no. 7024, pp. 425–430, 2005.

[4] Alvarez-Vasquez, F., Sims, K. J., Hannun, Y. A., and Voit, E. O., "Integration of kinetic information on yeast sphingolipid metabolism in dynamical pathway models," *Journal of Theoretical Biology*, vol. 226, no. 3, pp. 265–291, 2004.

[5] Bader, G. and Deuflhard, P., "A semi-implicit midpoint rule for stiff systems of ordinary differential-equations," *Numerische Mathematik*, vol. 41, no. 3, pp. 373–398, 1983.

[6] Bailey, J. E., "Toward a science of metabolic engineering," *Science*, vol. 252, no. 5013, pp. 1668–1675, 1991.

[7] Bailey, J. E., "Lessons from metabolic engineering for functional genomics and drug discovery," *Nature Biotechnology*, vol. 17, no. 7, pp. 616–618, 1999.

[8] Bates, D. M. and Watts, D. G., *Nonlinear regression analysis and its applications*. Wiley series in probability and mathematical statistics. Applied probability and statistics, New York: Wiley, 1988.

[9] Borisuk, M. T. and Tyson, J. J., "Bifurcation analysis of a model of mitotic control in frog eggs," *Journal of Theoretical Biology*, vol. 195, no. 1, pp. 69–85, 1998.

[10] Cantu-Paz, E., *Efficient and accurate parallel genetic algorithms*. Genetic algorithms and evolutionary computation; 1, Boston, Mass.: Kluwer Academic Publishers, 2000.

[11] Carlson, J. M. and Doyle, J., "Highly optimized tolerance: A mechanism for power laws in designed systems," *Physical Review E*, vol. 60, no. 2, pp. 1412–1427, 1999.

[12] Carlson, J. M. and Doyle, J., "Complexity and robustness," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 99, pp. 2538–2545, 2002.

[13] Cascante, M., Boros, L. G., Comin-Anduix, B., de Atauri, P., Centelles, J. J., and Lee, P. W. N., "Metabolic control analysis in drug discovery and disease," *Nature Biotechnology*, vol. 20, no. 3, pp. 243–249, 2002.

[14] Cash, J. R. and Karp, A. H., "A variable order runge-kutta method for initial-value problems with rapidly varying right-hand sides," *Acm Transactions on Mathematical Software*, vol. 16, no. 3, pp. 201–222, 1990.

[15] Clausing, P., "Regarding the absorption time and its measurement through a current experiment," *Annalen Der Physik*, vol. 7, no. 4, pp. 489–520, 1930.

[16] Coleman, T. F. and Li, Y. Y., "An interior trust region approach for nonlinear minimization subject to bounds," *Siam Journal on Optimization*, vol. 6, no. 2, pp. 418–445, 1996.

[17] Cornish-Bowden, A. and Cardenas, M. L., "Complex networks of interactions connect genes to phenotypes," *Trends in Biochemical Sciences*, vol. 26, no. 8, pp. 463–465, 2001.

[18] Cornish-Bowden, A. and Cardenas, M. L., "Systems biology: Metabolic balance sheets," *Nature*, vol. 420, no. 6912, pp. 129–130, 2002.

[19] Cornish-Bowden, A., *Analysis of enzyme kinetic data*. Oxford; New York: Oxford University Press, 1995.

[20] Dandekar, T., Schuster, S., Snel, B., Huynen, M., and Bork, P., "Pathway alignment: application to the comparative analysis of glycolytic enzymes," *Biochemical Journal*, vol. 343, pp. 115–124, 1999.

[21] Edwards, J. S., Ibarra, R. U., and Palsson, B. O., "In silico predictions of escherichia coli metabolic capabilities are consistent with experimental data," *Nature Biotechnology*, vol. 19, no. 2, pp. 125–130, 2001.

[22] Edwards, J. S. and Palsson, B. O., "The escherichia coli mg1655 in silico metabolic genotype: Its definition, characteristics, and capabilities," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 97, no. 10, pp. 5528–5533, 2000.

[23] Enright, W. H. and Hull, T. E., "Test-results on initial value methods for non-stiff ordinary differential equations," *Siam Journal on Numerical Analysis*, vol. 13, no. 6, pp. 944–961, 1976.

[24] Field, R. J. and Noyes, R. M., "Oscillations in chemical systems.4. limit cycle behavior in a model of a real chemical-reaction," *Journal of Chemical Physics*, vol. 60, no. 5, pp. 1877–1884, 1974.

[25] Fitzhugh, R., "Thresholds and plateaus in the hodgkin-huxley nerve equations," *Journal of General Physiology*, vol. 43, no. 5, pp. 867–896, 1960.

[26] Gear, C. W., "Simultaneous numerical solution of differential-algebraic equations," *Ieee Transactions on Circuit Theory*, vol. CT18, no. 1, pp. 89–, 1971.

[27] Gibbs, J. B., "Mechanism-based target identification and drug discovery in cancer research," *Science*, vol. 287, no. 5460, pp. 1969–1973, 2000.

[28] Goel, G. and Voit, E. O., "Bichemical systems toolbox: User's guide," 2006.

[29] Goldberg, D. E., *Genetic algorithms in search, optimization, and machine learning.* Reading, Mass.: Addison-Wesley Pub. Co., 1989.

[30] Gottwald, B. A. and Wanner, G., "Comparison of numerical-methods for stiff differential-equations in biology and chemistry," *Simulation*, vol. 38, no. 2, pp. 61–66, 1982.

[31] Hahn, W. C. and Weinberg, R. A., "Modelling the molecular circuitry of cancer," *Nature Reviews Cancer*, vol. 2, no. 5, pp. 331–341, 2002.

[32] Hairer, E., Nrsett, S. P., and Wanner, G., *Solving ordinary differential equations.* Springer series in computational mathematics; 8, 14, Berlin; New York: Springer-Verlag, 2nd rev. ed., 1993.

[33] Hendriks, B. S., Opresko, L. K., Wiley, H. S., and Lauffenburger, D., "Quantitative analysis of her2-mediated effects on her2 and epidermal growth factor receptor endocytosis - distribution of homo- and heterodimers depends on relative her2 levels," *Journal of Biological Chemistry*, vol. 278, no. 26, pp. 23343–23351, 2003.

[34] Henning, P., Merrill, A. H., and Wang, M. D., "Dynamic pathway modeling of sphingolipid metabolism," in *Proceedings of the 26th Annual International Conference of the IEEE EMBS*, (San Francisco, CA, USA), pp. 2913–2916, IEEE, 2004.

[35] Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., and Woodward, C. S., "Sundials: Suite of nonlinear and differential/algebraic equation solvers," *Acm Transactions on Mathematical Software*, vol. 31, no. 3, pp. 363–396, 2005.

[36] Holland, J. H., *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* Ann Arbor: University of Michigan Press, 1975.

[37] Hull, T. E., Enright, W. H., Fellen, B. M., and Sedgwick, A. E., "Comparing numerical methods for ordinary differential equations," *Siam Journal on Numerical Analysis*, vol. 9, no. 4, pp. 603–637, 1972.

[38] Irvine, D. H. and Savageau, M. A., "Efficient solution of nonlinear ordinary differential-equations expressed in s-system canonical form," *Siam Journal on Numerical Analysis*, vol. 27, no. 3, pp. 704–735, 1990.

[39] Iserles, A., *A first course in the numerical analysis of differential equations.* Cambridge; New York: Cambridge University Press, 1996.

[40] Jorba, A. and Zou, M. R., "A software package for the numerical integration of odes by means of high-order taylor methods," *Experimental Mathematics*, vol. 14, no. 1, pp. 99–117, 2005.

[41] Kalivas, J. H., *Adaption of simulated annealing to chemical optimization problems.* Data handling in science and technology; v. 15, Amsterdam; New York: Elsevier, 1995.

[42] Kee, R. J., Coltrin, M. E., and Glarborg, P., *Chemically reacting flow: theory and practice.* New York: Wiley-Interscience, 2003.

[43] Kimura, S., Kawasaki, T., Hatakeyama, M., Naka, T., Konishi, F., and Konagaya, A., "Obiyagns: a grid-based biochemical simulator with a parameter estimator," *Bioinformatics*, vol. 20, no. 10, pp. 1646–1648, 2004.

[44] King, E. L. and Altman, C., "A schematic method of deriving the rate laws for enzyme-catalyzed reactions," *Journal of Physical Chemistry*, vol. 60, no. 10, pp. 1375–1378, 1956.

[45] Kitano, H., "Computational systems biology," *Nature*, vol. 420, no. 6912, pp. 206–210, 2002.

[46] Kitano, H., "Systems biology: A brief overview," *Science*, vol. 295, no. 5560, pp. 1662–1664, 2002.

[47] Kitano, H., "Tumor tactics," *Nature*, vol. 426, p. 125, 2003.

[48] Larter, R., Bush, C. L., Lonis, T. R., and Aguda, B. D., "Multiple steady-states, complex oscillations, and the devils staircase in the peroxidase-oxidase reaction," *Journal of Chemical Physics*, vol. 87, no. 10, pp. 5765–5771, 1987.

[49] Lauffenburger, D. A. and Linderman, J. J., *Receptors: models for binding, trafficking, and signalling*. New York: Oxford University Press, 1993.

[50] Lehninger, A. L., Nelson, D. L., and Cox, M. M., *Principles of biochemistry*. New York, NY: Worth Publishers, 2nd ed., 1993.

[51] Leskovac, V., *Comprehensive enzyme kinetics*. New York: Kluwer Academic/Plenum Pub., 2003.

[52] Levin, E., "Grand challenges to computational science," *Communications of the Acm*, vol. 32, no. 12, pp. 1456–1457, 1989.

[53] Mathews, J. H. and Fink, K. D., *Numerical methods using MATLAB*. Upper Saddle River, N.J.: Prentice Hall, 3rd ed., 1999.

[54] Mavrovouniotis, M., Stephanopoulos, G., and Stephanopoulos, G., "Synthesis of biochemical production routes," *Computers Chemical Engineering*, vol. 16, no. 6, pp. 605–619, 1992.

[55] Mavrovouniotis, M. L., "Synthesis of reaction-mechanisms consisting of reversible and irreversible steps.2. formalization and analysis of the synthesis algorithm," *Industrial and Engineering Chemistry Research*, vol. 31, no. 7, pp. 1637–1653, 1992.

[56] Mavrovouniotis, M. L. and Chang, S., "Hierarchical neural networks," *Computers and Chemical Engineering*, vol. 16, no. 4, pp. 347–369, 1992.

[57] Mavrovouniotis, M. L. and Stephanopoulos, G., "Synthesis of reaction-mechanisms consisting of reversible and irreversible steps.1. a synthesis approach in the context of simple examples," *Industrial and Engineering Chemistry Research*, vol. 31, no. 7, pp. 1625–1637, 1992.

[58] Mavrovouniotis, M. L., Stephanopoulos, G., and Stephanopoulos, G., "Computer-aided synthesis of biochemical pathways," *Biotechnology and Bioengineering*, vol. 36, no. 11, pp. 1119–1132, 1990.

[59] Mavrovouniotis, M. L., Stephanopoulos, G., and Stephanopoulos, G., "Estimation of upper-bounds for the rates of enzymatic-reactions," *Chemical Engineering Communications*, vol. 93, pp. 211–236, 1990.

[60] Mendes, P., "Biochemistry by numbers: simulation of biochemical pathways with gepasi 3," *Trends in Biochemical Sciences*, vol. 22, no. 9, pp. 361–363, 1997.

[61] Merrill, A. H., "De novo sphingolipid biosynthesis: A necessary, but dangerous, pathway.," *Journal of Biological Chemistry*, vol. 277, no. 29, pp. 25843–25846, 2002.

[62] Merrill, A. H., Sullards, M. C., Allegood, J. C., Kelly, S., and Wang, E., "Sphingolipidomics: High-throughput, structure-specific, and quantitative analysis of sphingolipids by liquid chromatography tandem mass spectrometry," *Methods*, vol. 36, no. 2, pp. 207–224, 2005.

[63] Mitchell, M., *An introduction to genetic algorithms*. Complex adaptive systems, Cambridge, Mass.: MIT Press, 1996.

[64] Nelder, J. and Mead, R., "A simplex method for function minimization," *Computer Journal*, vol. 7, pp. 308–313, 1965.

[65] Ni, T. C. and Savageau, M. A., "Application of biochemical systems theory to metabolism in human red blood cells - signal propagation and accuracy of representation," *Journal of Biological Chemistry*, vol. 271, no. 14, pp. 7927–7941, 1996.

[66] Ni, T. C. and Savageau, M. A., "Model assessment and refinement using strategies from biochemical systems theory: Application to metabolism in human red blood cells," *Journal of Theoretical Biology*, vol. 179, no. 4, pp. 329–368, 1996.

[67] Noble, D., "Modeling the heart - from genes to cells to the whole organ," *Science*, vol. 295, no. 5560, pp. 1678–1682, 2002.

[68] Otten, R. H. J. M. and Ginneken, L. P. P. P. v., *The annealing algorithm*. Boston: Kluwer Academic Publishers, 1989.

[69] Petzold, L., "Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential-equations," *Siam Journal on Scientific and Statistical Computing*, vol. 4, no. 1, pp. 136–148, 1983.

[70] Plowman, K. M., *Enzyme kinetics*. McGraw-Hill series in advanced chemistry, New York,: McGraw-Hill, 1971.

[71] Poulin, P. and Theil, F. P., "Prediction of pharmacokinetics prior to in vivo studies. ii. generic physiologically based pharmacokinetic models of drug disposition," *Journal of Pharmaceutical Sciences*, vol. 91, no. 5, pp. 1358–1370, 2002.

[72] Press, W. H., *Numerical recipes in C++: the art of scientific computing*. Cambridge, UK; New York: Cambridge University Press, 2nd ed., 2002.

[73] Quinn, M. J., *Parallel programming in C with MPI and OpenMP*. Boston: McGraw-Hill Higher Education, 2004.

[74] Reed, J. L., Vo, T. D., Schilling, C. H., and Palsson, B. O., "An expanded genome-scale model of escherichia coli k-12 (ijr904 gsm/gpr)," *Genome Biology*, vol. 4, no. 9, pp. –, 2003.

[75] Richaud, C., Menginlecreulx, D., Pochet, S., Johnson, E. J., Cohen, G. N., and Marliere, P., "Directed evolution of biosynthetic pathways - recruitment of cysteine thioethers for constructing the cell-wall of escherichia-coli," *Journal of Biological Chemistry*, vol. 268, no. 36, pp. 26827–26835, 1993.

[76] Rusling, J. F. and Kumosinski, T. F., *Nonlinear computer modeling of chemical and biochemical data*. San Diego: Academic Press, 1996.

[77] Sander, C., "Genomic medicine and the future of health care," *Science*, vol. 287, no. 5460, pp. 1977–1978, 2000.

[78] Schafer, E., "New approach to explain high-irradiance-responses of photomorphogenesis on basis of phytochrome," *Journal of Mathematical Biology*, vol. 2, no. 1, pp. 41–56, 1975.

[79] Schena, M., Shalon, D., Davis, R. W., and Brown, P. O., "Quantitative monitoring of gene-expression patterns with a complementary-dna microarray," *Science*, vol. 270, no. 5235, pp. 467–470, 1995.

[80] Schilling, C. H. and Palsson, B. O., "The underlying pathway structure of biochemical reaction networks," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 95, no. 8, pp. 4193–4198, 1998.

[81] Schilling, R. J. and Harris, S. L., *Applied numerical methods for engineers using MATLAB and C*. Pacific Grove, CA: Brooks/Cole, 2000.

[82] Schuster, S., Dandekar, T., and Fell, D. A., "Detection of elementary flux modes in biochemical networks: a promising tool for pathway analysis and metabolic engineering," *Trends in Biotechnology*, vol. 17, no. 2, pp. 53–60, 1999.

[83] Schuster, S., Hilgetag, C., Woods, J. H., and Fell, D. A., "Reaction routes in biochemical reaction systems: Algebraic properties, validated calculation procedure and example from nucleotide metabolism," *Journal of Mathematical Biology*, vol. 45, no. 2, pp. 153–181, 2002.

[84] Shampine, L. F., "Implementation of rosenbrock methods," *Acm Transactions on Mathematical Software*, vol. 8, no. 2, pp. 93–113, 1982.

[85] Shampine, L. F. and Reichelt, M. W., "The matlab ode suite," *Siam Journal on Scientific Computing*, vol. 18, no. 1, pp. 1–22, 1997.

[86] Shampine, L. F., Watts, H. A., and Davenport, S. M., "Solving nonstiff ordinary differential equations - state of art," *Siam Review*, vol. 18, no. 3, pp. 376–411, 1976.

[87] Spiegel, S. and Merrill, A. H., "Sphingolipid metabolism and cell growth regulation," *Faseb Journal*, vol. 10, no. 12, pp. 1388–1397, 1996.

[88] Stelling, J., Klamt, S., Bettenbrock, K., Schuster, S., and Gilles, E. D., "Metabolic network structure determines key aspects of functionality and regulation," *Nature*, vol. 420, no. 6912, pp. 190–193, 2002.

[89] Stephanopoulos, G. and Vallino, J. J., "Network rigidity and metabolic engineering in metabolite overproduction," *Science*, vol. 252, no. 5013, pp. 1675–1681, 1991.

[90] Stoer, J. and Bulirsch, R., *Introduction to numerical analysis.* New York: Springer-Verlag, 2nd ed., 1992.

[91] Tomita, M., Hashimoto, K., Takahashi, K., Shimizu, T. S., Matsuzaki, Y., Miyoshi, F., Saito, K., Tanida, S., Yugi, K., Venter, J. C., and Hutchison, C. A., "E-cell: software environment for whole-cell simulation," *Bioinformatics*, vol. 15, no. 1, pp. 72–84, 1999.

[92] Voit, E. O., "Metabolic modeling: a tool of drug discovery in the post-genomic era," *Drug Discovery Today*, vol. 7, no. 11, pp. 621–628, 2002.

[93] Voit, E. O., *Canonical nonlinear modeling: S-system approach to understanding complexity edited by.* New York: Van Nostrand Reinhold, 1991.

[94] Voit, E. O., *Computational analysis of biochemical systems: a practical guide for biochemists and molecular biologists.* New York: Cambridge University Press, 2000.

[95] Voit, E. O., "Models-of-data and models-of-processes in the post-genomic era," *Mathematical Biosciences*, vol. 180, no. 1-2, pp. 263–274, 2002.

[96] Wolkenhauer, O., Sreenath, S. N., Wellstead, P., Ullah, M., and Cho, K. H., "A systems- and signal-oriented approach to intracellular dynamics," *Biochemical Society Transactions*, vol. 33, pp. 507–515, 2005.

[97] Wolkenhauer, O., Ullah, M., Wellstead, P., and Cho, K. H., "The dynamic systems approach to control and regulation of intracellular networks," *Febs Letters*, vol. 579, no. 8, pp. 1846–1853, 2005.

[98] Woolf, P. J., Prudhomme, W., Daheron, L., Daley, G. Q., and Lauffenburger, D. A., "Bayesian analysis of signaling networks governing embryonic stem cell fate decisions," *Bioinformatics*, vol. 21, no. 6, pp. 741–753, 2005.

[99] Yamazaki, I., Yokota, K., and Nakajima, R., "Oscillatory oxidations of reduced pyridine nucleotide by peroxidase," *Biochemical and Biophysical Research Communications*, vol. 21, no. 6, p. 582, 1965.

[100] Yi, T. M., Huang, Y., Simon, M. I., and Doyle, J., "Robust perfect adaptation in bacterial chemotaxis through integral feedback control," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 97, no. 9, pp. 4649–4653, 2000.