SIMULATION OF ELECTROCHEMICAL AND STOCHASTIC SYSTEMS USING JUST IN TIME COMPILED DECLARATIVE LANGUAGES

Endre T. Somogyi

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Physics,
Indiana University
December 2014

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Peter J. Ortoleva

John M. Beggs

Randall Bramley

Sima Setayeshgar

Defense Date

November 7, 2014

ACKNOWLEDGMENTS

I thank my advisor, Dr. Peter Ortoleva for allowing me work on the electro-metabolism project which started this entire endeavor. I would like to personally thank Dr. James Glazier for his help in organizing my thesis defense. I thank Dr. Herbert Sauro for many useful discussions on chemical kinetics. I thank Dr. Randall Bramley for teaching me what I know about parallel algorithms and for his insight in numerical algorithms and algorithmic differentiation which begins the next chapter of this work. Finally I would like to thank my wife, Marie first for encouraging me to pursue a graduate career, and for her understanding and her indispensable help in creating this document.

I extend my thanks to the agencies whose financial support made this research possible: The Biocomplexity Institute, Indiana University, under grant NIH/NIGMS, R01 GM077138; Indiana University Departments of Physics and Chemistry for financial support, and University of Washington, under grant NIH/NIGMS, R01 GM081070.

Endre T. Somogyi

SIMULATION OF ELECTROCHEMICAL AND STOCHASTIC SYSTEMS USING JUST IN TIME COMPILED DECLARATIVE LANGUAGES

Computational biology is a relatively new discipline which sits at the intersection of computer science, physics, chemistry and biology. One of the primary goals of computational biology is to develop predictive algorithmic and mathematical models of biological processes.

A model description in a declarative language such as SBML express the structure of the model without having the specify the explicit control flow. Declarative descriptions have numerous advantages over lower level programming languages. The SBML language is specifically oriented towards describing biochemical systems: one simply has to list reactants and the relationships, as opposed to lower level procedural languages, where one would have to explicitly specify the computational details. Models may also be readily exchanged and reused in a variety of applications.

A number of interpreters exist for simulating SBML models, but to our knowledge, there are no Just In Time (JIT) compilers. Compiled languages often offer hundred fold performance improvements over interpreters. As simulations of cellular systems become more complex, particularly in multicellular models, the need for reusable and high performance simulation engines is becoming clear.

This thesis will describe the SBML JIT compiler, simulation and analysis library that was developed. The library has been designed to be extensible and offers superior performance to standard desktop simulators and supports a variety of analyses including time course simulation and a wide range of analysis features such as steady state and metabolic control analysis.

The library was used to develop a physically based model of the first component of the mitochondrial electron transport chain. Mitochondria play a essential role in cellular biology, and there exists a need for a physically accurate and interchangeable models of mitochondrial processes. Additionally, to demonstrate the stochastic capabilities of the library, a stochastic bistable chemical system was modeled and analyzed.

Peter J. Ortoleva

John M. Beggs

Randall Bramley

Sima Setayeshgar

Contents

1 Introduction **1**

1.1 Interoperability and Heterogeneous Models 4

1.2 Declarative Model Specification 6

1.3 Software Architecture 8

1.4 Reusable Mitochondrial Model 9

1.5 Outline 10

2 Dynamic Compilation of Declarative Languages for Systems Biology **12**

2.1 Introduction 12

2.2 Overview 13

2.2.1 Systems Biology Markup Language 13

2.3 Architecture 22

2.3.1 Design Goals 22

2.3.2 Component Based Design 24

2.4 SBML Language Compilation 26

2.4.1 Overview 28

2.4.2 Compiler Design Overview and LLVM Details 28

2.4.3 Implementation of SBML Language features 30

2.5 Results 37

2.5.1 Performance 37

2.5.2 Analysis features 43

2.5.3 Conserved Quantities 45

2.5.4 Spatial Systems using RoadRunner 51

2.6 Community Integration 56

3 Electrochemical Mitochondrial Modeling 57

3.1 Introduction 57

3.2 Mitochondrion Structure and Function 59

3.2.1 Electron Transport Chain 60

3.3 Quantitative Model Formalism 62

3.3.1 Reaction Rate Kinetics 63

3.3.2 Interpretation of Experimental Data 72

3.4 Spatial Electro-Chemical Kinetics 79

3.4.1 Electrical Double Layer 79

3.4.2 Physical Description 80

3.4.3 Non-Dimensionalization 87

3.5 Electrical Double Layer 89

3.5.1 Electrical Double Layer Models 89

3.5.2 Electric field from the charged compartments 90

3.5.3 Double Layer Capacitance Results 94

3.6 Conclusions 96

4 Stochastic Bi-Stability Analysis 98

4.1 Introduction 98

4.2 A Bistable System 99

4.3 Deterministic Approach 100

4.3.1 Deterministic Analysis 100

4.4 Stochastic Approach 101

4.4.1 Master Equation 102

4.4.2 Stochastic Analysis 106

4.4.3 Stochastic Simulation and Gillespie's Algorithm 108

4.4.4 Results 113

4.4.5 Detrended Fluctuation Analysis 119

4.5 Lattice Lotka Volterra 121

4.6 Conclusions 122

5 Conclusions and Future Work **123**

5.1 Jacobian Automatic Differentiation 124

5.2 Mutable Conserved Moieties 124

5.3 Better Steady State Solvers 124

5.4 Different Integrators 125

5.5 Script extension to use standard scripting languages 125

Appendix A Acronyms **127**

Bibliography **128**

Curriculum Vitae

Chapter 1

Introduction

A constructionist methodology begins with a known set of building blocks and attempts to assemble them into a larger, more complex system. This is the approach that an engineer or condensed matter physicist would take to build or understand a complex system. There is no better way to understand a system than to build it. The fundamental goal here is not discovering the underlying quantum-mechanical laws, but rather understanding how complex systems composed of simpler components behave and interact – in understanding the new laws that *emerge* when these building blocks interact. The constructionist perspective is complementary to the reductionist perspective that particle physicist would take. Here, the goal is to look ever lower and lower to finally arrive at the ultimate building blocks. Originally, these blocks were the atoms. Then it was discovered that atoms were made of electrons, protons and neutrons. These were then discovered to be made of quarks, and these in turn are perhaps made of strings.

There can be little doubt of the tremendous advancements in science that a reductionist perspective has had, however such a perspective may not be the most useful way of approaching the complex systems such as seen in biology or condensed matter physics. As Paul Anderson said, “The ability to reduce everything to simple fundamental laws does not imply the ability to start from those laws and reconstruct the universe” [4]. Knowing the exact properties of quarks in excruciating detail is not terribly helpful in understanding how a biological cell functions. Quantum chromo-dynamics simulations of quarks can take a months of super computer time [22]. Similarly with quantum-mechanical calculations, although these may be useful for calibrating larger scale molecular dynamics models. Molecular dynamics on the other hand start with the building blocks of amino acids and molecules and these are useful for calibrating models of larger scale components such as, say, ion pumps. A cell however contains millions if not billions of ion pumps and these pumps are constantly reacting with a large collection of reactants. A choice of quarks or elementary particles is not an appropriate choice as the building blocks of a cellular model, much as grains of sand would not be an appropriate choice of building block with which to construct the great pyramids. Because of their appropriate size and well defined behavior, the set of amino acids is an appropriate building block to construct protein models. The important point here is to choose the appropriate building blocks.

The software library developed in this thesis was developed as a tool for biophysics research and facilitates the construction, simulation and analysis of complex biological models that are composed of a set of potentially heterogeneous simpler models or building blocks. The development of tools and techniques has always gone hand in hand with the progression of scientific understanding. Hans Lippershey, an obscure spectacle-maker from Middleburg Zeeland – who was referred to as an "illiterate mechanick" by Huygens [48] – is generally credited with the invention of the telescope in 1608. Yet this illiterate mechanick's invention was used by the likes of Galileo, Kepler, Newton and Huygens himself to collect data and bring about a revolution in our understanding of the cosmos and physics. New tools have allowed the study of systems of ever decreasing as well as increasing size. The telescope allowed the observation of larger systems such as our solar system. The development of refrigeration techniques has allowed the study of smaller and smaller quantum mechanical systems.

The development of advanced refrigeration techniques in the industrial revolution of late nineteenth century allowed researchers such as Heike Kamerlingh Onnes to super-cool materials to near absolute zero, resulting in the discovery of super conductivity. This experimental observation lacked a theoretical explanation for thirty years [4].

Possibly one of the greatest innovations of the industrial revolution is the notion of interchangeable parts. Before the 1800's nearly all mechanisms were bespoke – each was typically custom built for a specific purpose, and one could not take a component from one mechanism and readily use it in another. Each building block here was custom made, from screws to springs to gears. One could not build a mechanism without first building the building blocks.

Much as the industrial revolution brought about radical change in the 1800's, the computer revolution has and is bringing about radical changes in how science is performed today. This revolution is rapidly opening up new avenues of scientific advancement. The computational hardware resources available today are allowing scientists of all fields to develop larger and more sophisticated simulations and analyze ever-increasing data sets.

The computer revolution would not have been possible without the notion of interchangeable parts. Engineers have long since known of the benefits of component interoperability. When an engineer designs a circuit board or a mechanism, these systems are constructed out of a number of standard building blocks. Boards are constructed of a set of integrated circuit (IC) elements. These in turn are constructed of basic elements such as transistors, resistors, etc. These IC elements have standardized packages and interfaces which allow them to be assembled to form more complex systems. It is a tremendous advantage to be able to browse a catalog and choose a set of blocks to develop a novel component by connecting them. The notion of interoperability is commonly found in nature itself as evidenced by the great many numbers of conserved gene circuits and protein complexes that are found in a large number of species.

This notion of interoperability is not as prevalent as it could be in the areas of biological modeling and computer software. A computational model of physical phenomena is a mathematical representation meant to quantitatively describe the salient aspects of the biological system they represent, as well as provide insights and predict the system's dynamics in response to changing conditions. A mathematical model need not replicate every internal detail of the system it is describing. Models of simpler biological systems form the building blocks of more complex systems of biological models. This dissertation presents a software library that was developed to combine a potentially large set of heterogeneous biological models into a larger and potentially more complex system of models. This process of assembling simpler models into systems gives insights into how these larger and more complex systems form and function in nature.

In the constructionist or "bottom-up" modeling approach [3,14,36], a multitude of models may be connected to represent more complex higher level biological systems. However, guaranteeing the validity and predictability of the compounded ensemble may become increasingly challenging as more components are integrated. Such modeling platforms may combine very heterogeneous models from different scales, ranging from bimolecular detail to sub-cellular networks to the continuum spatial level. At the molecular level, such platforms contain a large number of kinetic models (e.g., ionotropic receptors, channels and exchangers) as well as complex second-messenger pathways, all integrated at the cellular level into morphologically detailed neuronal models. As a result, such platforms contain an overwhelmingly large number of models and parameters. Each of the models present in the platform must be independently calibrated and validated with respect to experimental results.

As more experimental data becomes available, as our understanding of biochemical interactions increases, so will the size and complexity of biochemical models. Not only is the size of chemical network models increasing, but so is the complexity and sophistication of these models' rate laws and reaction dynamics. These complex networks contain an ever-increasing number of parameters that must be fitted from experimental data. Chemical networks are also being used to model the sub-cellular reaction network in a variety of cellular and virtual tissue simulators. Thus, there is a need for a modular, high-performance simulation library that can excel at simulating and analyzing such increasingly complex models.

Currently, the only way to determine the values for such model parameters is by simulating the models and comparing the simulation results to experimental data. This entails a very large number of simulations and consumes a significant amount of computational time and resources. The Systems Biology Markup Language (SBML) simulation library developed here enables the development of such complex biological models [15]. No other SBML simulation engine has the level of performance or interoperability as the one developed here.

1.1 Interoperability and Heterogeneous Models

For a mathematical model to be useful, not only must it provide some predictive, or at least explanatory capability, but it must also be *interchangeable* among different platforms and programs. For many years (and very sadly, even still) mathematical models were distributed as FORTRAN programs. These programs typically have some proprietary input and output file formats. The most common way to re-use these programs is to attempt to compile them oneself, then write a generator/parser to create input files and parse output files, and then (of course) a script to actually execute the program.

Programmatically interfacing with external programs (assuming one can even compile them) is fraught with error primarily due to the large amount of state that these programs may require to be present on the file system. These programs are typically written under the assumption that they will be run once, for one particular calculation and then discarded. This presents a difficulty for calling programs, as they must determine the current state of the file system, copy and convert large portions of internal state into input files, call these programs, parse the output, and finally clean up after these programs are run.

Perhaps worse still are models published in a proprietary format such as MATLAB. This requires that the user also purchase and use MATLAB which may be a significant financial outlay. Worse even still are models published only as equations in a journal. These models are typically missing terms, missing parameters, or use unusual mathematical conventions. Thus there is a need for mathematical models published in a self-contained reusable interchange format.

Cellular and virtual tissue simulators such as CompuCell3D [75] or V-Cell [70], or CELL-SZYZS [43] operate on complex spatial cellular and virtual tissue domains which have much longer time and length scales than the typical sub-cellular reaction network. The sub-cellular reaction network is however still kept in inter-cellular domains such as the cytoplasm and these reaction networks are typically represented as SBML models. Thus, there is a need for software components that can easily be used as a component of existing cellular simulators.

Many of these simulators have used SOSLib [56]. This is an older simulation library that has the benefits of being fairly small and self contained, however it does not appear to have been maintained in some time and has a very limited Application programming interface (API). SOSLib is also very difficult to build on modern operating systems as it relies on a number of older versions of software packages which are difficult to build on modern operating systems. Some virtual tissue simulators such as CompuCell3D have previously used SOSLib but have already switched to the libRoadRunner library developed here.

The problems encountered in biology tend to be much more heterogeneous than those in condensed matter physics. In condensed matter physics, one frequently deals with a ensembles of homogeneous particles such as a gas or liquid composed of one chemical species. Highly complex behavior can arise from a collection of simple fluid particles; even a collection of particles interacting with only a Van Der Walls potential is sufficient to reproduce most of the behaviors encountered in Newtonian fluid dynamics. Biological building blocks tend to be more complex, and they tend to be heterogeneous. For example, a cell has a set of internal reactions which may be modeled via a reaction-kinetics model (such as the ones which will be described in this thesis). Cells are typically found in a aqueous environment which allows free diffusion of reactants to and from the cell. This environment is most efficiently treated as a continuum fluid model. The library developed here has been used in such simulations, for example, spatial models of complex biological processes such as glutameric synapse simulations [3, 36]. These simulations use a chemical network model to simulate the reactions and distributions of compounds in the synaptic cleft. These simulations also contain a large number of free parameters that can only be determined by running a large number of simulations, hence the need for a high performance simulation library.

Another example of such a heterogeneous system is the growth of tumor cells. Here again, the internal cell cycle is modeled with a reaction-kinetics network, and the cells exist in a fluid environment which supports free diffusion of reactants such as oxygen. A time series of this simulation, developed by Powathil, Chaplain and Swat [62] is shown in fig. 1.1.

1.2 Declarative Model Specification

As mentioned in 1.1, many biological models are published in FORTRAN or MATLAB format. MATLAB is an excellent environment for developing a numerical algorithm or performing data analysis; however, it may not be the ideal environment to develop and publish a biological model.

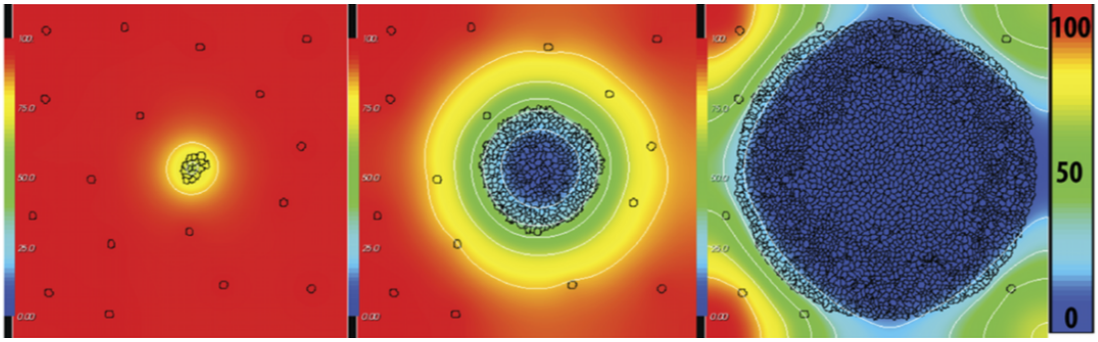


Figure 1.1: A heterogeneous model of the growth of cancer tumor cells from [62]. Here, the internal cell cycle dynamics is modeled as a reaction-kinetics network and this network is coupled continuum fluid model which supports free diffusion. The background field represents oxygen concentration.

MATLAB, FORTRAN, and to an extent, C are all what are called *procedural* languages. In a procedural language, the exact details of the computation are encoded in a sequence of language statements. To define a biological simulation in a procedural language, one must first identify all the state variables, store these in a state vector, write a series of routines which calculate the rate of change of the state vector and finally explicitly specify all the exact details of calling an integrator.

Biological reaction-kinetics models however can be much simpler. At a minimum, all one should be required to do is list a set of reactants and products, and list a set of reactions that they participate in. It can be up to the language runtime to determine low level details such as the exact layout of the state vector and what the actual low-level differential equations are that must be solved. In a declarative language, the user never needs to be concerned with these low-level details.

This is exactly what the declarative language, Systems Biology Markup Language (SBML), allows users to accomplish – define a biological model by specifying only a list of reactants and products, and a list of reactions. Users can optionally specify a list of rules or relationships the parameters or reactants must abide by as well. Therefore, SBML can be thought of as more of a *declarative* rather than a procedural language. In a declarative language, the programmer must explicitly define the exact computation to be performed. The compiler of a declarative language must infer semantic meaning from a set of potentially disjointed pieces of information. In a procedural language, however, the onus is on the programmer to explicitly define the exact computation to be performed. The task of compiling a procedural language is typically much simpler than compiling a declarative language, as a procedural language places more of the burden for explicitly defining the computation on the programmer rather than the compiler.

One of the primary advantages of using a declarative format to specify chemical models is its interoperability with a wide range of existing software. Much like (standard) Hypertext Markup Language (HTML), SBML can be displayed on a very wide range of platforms ranging from handheld phones like iOS and Android devices all the way up to desktop computers, and a variety of operating systems such as OSX, Linux, FreeBSD, Solaris, and evidently even Windows can display web pages encoded in HTML. SBML is similar in this respect to HTML, as it is designed as an interchange format.

A wide range of software exists for graphically building SBML models. At the time of writing, 263 different software packages can import or export SBML models. These are listed in the SBML software guide at [67]. A number of graphical model builders such as Cell Designer [19], COPASI [44], and a variety of other tools can graphically display and allow users to create and manipulate SBML models. Using such graphical builders, users do not even have to write a single line of code.

Specifying the model in a higher level, declarative format such as SBML allows software tools to perform a wide range of analyses (such as stoichiometric analysis of the model) which would be exceedingly difficult to perform if the model were explicitly specified in a low level procedural format. Declarative languages are ideal for such tasks as biological model specification, although it may not be appropriate for all circumstances: one would not want to write applications or operating systems in declarative languages. In such problem domains, one does not need to explicitly specify the exact computation or the exact layout of memory blocks, or other low level minutiæ.

1.3 Software Architecture

The LibRoadRunner library developed here is a self-contained, cross-platform library which is designed to have a small memory footprint. There were three primary architectural goals:

interoperability, modularity and speed. The library is designed to be used in existing virtual tissue simulators, hence all internal code is C++ but the library has extensive language bindings to Python as well as the native C++ interface. All internal modules are written as a set of loosely coupled components. This creates a modular framework which readily allows the development of new components such as integrators, steady state solvers or back ends. The most novel feature of the LibRoadRunner library is this is the first known SBML simulation engine which supports Just In Time Compile (JIT) compilation. Briefly, this is a technique where a model described in a source language (SBML in this case) is directly compiled to native executable machine code in memory. The library developed here appears to be the first SBML simulation engine that achieves linear scaling 2.5.1. The other simulation engines that were benchmarked could at best achieve quadratic scaling, and at worst achieve only exponential scaling. The library also provides a rich suite of analysis capabilities (MCA, stoichiometric, etc.).

The need for interoperability extends beyond just a library which is a small, self-contained component – it must also be usable from a variety of different languages. Even though most cellular and virtual tissue simulators are written in C++, many of them have a scripting language interface (typically in Python); some are also written in Java. C++ may be an ideal choice for time-critical components, however it is not a good choice for interactive use and rapid prototyping such as in an interactive MATLAB session. Therefore, the library developed here supports a variety of language bindings via SWIG [8]. The library currently supports a native C++ API as well as a high performance Python binding. A JavaScript binding for use in Node.js is currently under development.

1.4 Reusable Mitochondrial Model

The transduction of free energy from sugars to phosphorylate ADP to ATP is one of the most fundamental and important reaction pathways in biology. Without a constant supply of high free energy ATP, life as we know it would not exist. A great deal is known about this reaction network; however, no complete, interchangeable model of the glycolysis/oxidative phosphorylation reaction network appears to exist.

Selivanov, et al., have published a series of mitochondrial Electron Transport Chain (ETC) models [68, 69]. These however were hard-coded into a single monolithic C++ program which provided no programmatic interface or API. Effectively, this program is not exchangeable and cannot be practically used in any existing software due to its lack of an API.

1.5 Outline

In the biological modeling process, one first starts with empirical observations and combines these with the known laws of mathematics, physics and chemistry using human reason and insight to create a declarative, quantitative description or model of the phenomenon being investigated. Traditionally, one then typically writes down a set of differential equations that describe or govern the system. Then a computer program is written which implements these differential equations in an form that a machine can accept. An automated process then compiles this code into an executable form, combines it with various mathematical software libraries such as integrators, linear solvers, etc. The output of this program is then fed to an analysis package such as Mathematica or MATLAB. The biological modeling process is represented in fig. 1.2.

The first steps in the process fundamentally require human reason and insight, and for the foreseeable future, it is unlikely that any machine will be capable of accomplishing this task. Standard software packages have always existed for the latter steps such as excellent existing compilers, integrators, analysis packages, etc. This thesis focuses on developing the tools that enable the automation of the middle steps and combines the later 2/3 of the biological modeling process into a single, self-contained library that is designed to be used in a wide variety of applications.

In this thesis, Chapter 2 focuses on the LibRoadRunner library developed herein. This chapter gives an overview of declarative language specification of biological models, discusses the process of declarative language Just In Time compilation and discusses the internal architecture of the library in detail. This chapter also compares the library's performance benchmarks against a number of existing libraries'.

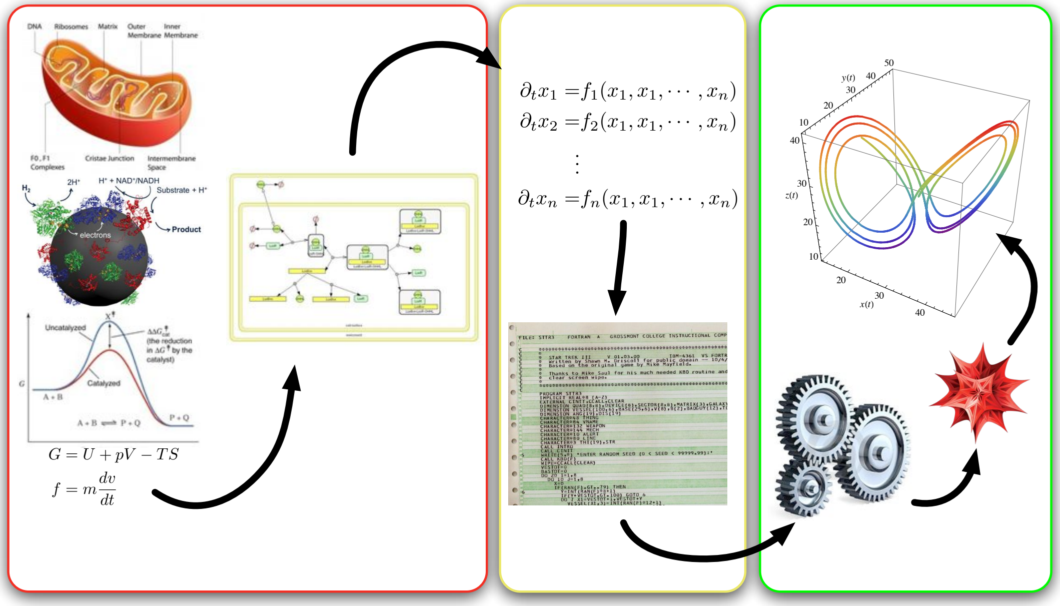


Figure 1.2: Overview of the biological modeling process. The section in red fundamentally requires human insight and will likely not be automatable in the near future. The section in yellow is often a manual process; this thesis will develop the tools to automate this section. The section in green has always been easily automated by computer applications.

Chapter 3 discuses the development of a reusable model of the first complex of the mitochondrial electron transport chain. The library developed here was used to perform the simulations. Chapter 4 will demonstrate the capabilities of the stochastic integrator that is part of the this library. This chapter also compares stochastic to deterministic models and will include a derivation of the master equation and the stochastic simulation algorithms which are required to perform such stochastic simulations.

Finally, Chapter 5 concludes with a number of planned future additions to the library.

Chapter 2

Dynamic Compilation of Declarative Languages for Systems Biology

2.1 Introduction

LibRoadRunner is an open-source, cross-platform library for the numerical analysis of cellular models expressed in Systems Biology Markup Language (SBML) [45]. The library supports a variety of analyses including time course simulation and steady state analysis. As simulations of cellular systems become more complex, particularly in multicellular models, the need for reusable and high performance simulation engines is becoming clear [75]. The LibRoadRunner library has been designed to be extensible and offers superior performance to standard desktop simulators which will be demonstrated in § 2.5.1. This chapter describes the architectural design of the LibRoadRunner library which has the following key attributes: (1) extensible modular architecture which readily allows the development of new components without altering any existing code (2) first known SBML JIT compiler, (3) capable and well documented native bindings for Python, and C++, (4) extensive but easy-to-use API which allows SBML model time series and steady state simulations, model introspection and editing and a host of analysis features, (5) a self-contained library for which we provide source code and binary packages for a variety of platforms, including 64 and 32 bit Linux (Intel), 64 bit OSX and 32 bit Windows. The code is written in platform independent C++ and a native Python binding is also provided.

LibRoadRunner (as of the time of this writing) is linked against libSBML 5.9, hence we support the entire SBML Level 3, Version 1 (L3V1) specification, with the exception of delay equations, algebraic rules or fast reactions.

2.2 Overview

2.2.1 Systems Biology Markup Language

Systems Biology Markup Language SBML [45] is a declarative representation format primarily oriented towards describing and communicating computational models of biological processes occurring in Continuous Well-Stirred Compartments (CWSC). A significant portion of phenomena in biology and chemistry and chemical engineering are commonly described as networks of reacting solutes in a set of such CWSCs. The CWSC approximation holds when the mass diffusion rate is greater than approximately 5 times the maximum chemical reaction rate, or the compartment is sufficiently small. More precisely, when $L^2 < J\tau$, where L is the characteristic length of the compartment, J is the mass diffusion flux, and τ is the characteristic time of system reactions rates (inverse reaction rate).

Such models are frequently used to describe a diverse range of biological processes such as cellular and sub-cellular processes, metabolic and reaction networks, cellular signaling pathways, regulatory networks and electrophysiology.

SBML was originally developed to model biochemical kinetics networks, but has since been augmented with an extensive event based programming model, auxiliary processes not defined by the reaction network and arbitrary mathematical functions. SBML is often misunderstood as being only a system for specifying ordinary differential equations (ODEs) as many of the earlier SBML publications focused on this area. These additional capabilities allow SBML to describe significantly more than just biochemical reaction kinetics models.

Although SBML can be used to describe nearly any computable process, as any computable process may be described in terms of continuous dynamical systems, this is not always practical. SBML is however ideal for representing biochemical models and has become the de facto standard in this problem domain.

Representing a model in SBML has numerous advantages over explicitly specifying a model in a comparatively low level programming language such as Python, MATLAB or C++. The SBML language is specifically oriented towards describing biochemical systems in a declarative manner. In SBML, one simply has to list reactants and the relationships among the reactants to define a model. In lower level procedural languages, one would have to explicitly specify the computation – explicitly specify constructs such as the stoichiometry matrix, state vectors, ODE, etc. and assemble these together with an integrator package.

The principal advantage for model specification in SBML is the interchangeability of models. In addition to LibRoadRunner, numerous SBML simulation engines exist, each oriented towards a specific niche: desktop end user applications such as COPASI [44] or Java simulation environments such as systems biology simulations core algorithm biology library. SBML models may be shared and published in exchange repositories such as the BioModels database [10]. Because SBML is a language neutral specification, a range of packages exist to edit, view or simulate models in a variety of different languages and environments. SBML is intended to ensure the availability of models well beyond the lifetime of the packages that originally created them.

The SBML language provides a set elements for building biochemical models which will be summarized here. All elements in SBML are named and are given a unique identifier (id) or symbol name, which is just a unique character string. Physical quantities are specified in SBML as chemical species, parameters, or compartments. Relationships and dynamics of these quantities are represented as rules or reactions.

Although SBML models are frequently solved deterministically, the SBML specification does not specify what method is used for time evolution. This allows implementations freedom to choose appropriate time evolution methods such as deterministic or stochastic integrators. SBML only defines the *state* of the system, it does not specify how that system should evolve in time. The LibRoadRunner library provides a number of different integrators which evolve the system over time. These are discussed in §2.3.2.

An SBML model defines a set of state variables which may be either parameters, chemical species or possibly compartments. The time evolution of chemical species is determined by the reaction network which is specified via a set of reactions. Any element in SBML may be defined by a rule; this defines the element as a mathematical relationship of other SBML elements. A model may have a set of events which define a discontinuous state change of any SBML element. A number of other elements which do not effect the model state such as unit definitions or constraints may also be specified. These will not be discussed here. (For further details, see the SBML specification at [46].) The key SBML elements which define the model state and affect the model dynamics follow.

Parameters

SBML defines three kinds of *symbols* which may be evaluated to yield a numerical value: parameters, compartments and species. Parameters are the most general kind in that they are not assigned any specific physical meaning. Parameters may be interpreted as any arbitrary numerical variable.

Species

Chemical species are divided into two categories, floating and boundary. Floating species may be produced or consumed in a reaction network and their value over time is determined by reaction kinetics. Boundary species are *not* produced or consumed in a reaction, and their value is often fixed over time since they act as boundary conditions. The value of a boundary species may however vary in time, as a boundary species may have an event which changes its value or it may have a *rate rule* which alters its value. For example, a boundary species may be considered an infinite source or sink for a chemical species. In this case, even though they participate in a reaction network as products or reactants, the value of the boundary species remains constant.

The state variables of an SBML model may consist of species, compartments or parameters. Species are intended to represent a chemical species, and these may be either floating or boundary species. There are slightly different semantics between floating and boundary species (for details, see [46]). The main difference is that the floating species value over time is typically determined by the chemical reaction network, whereas the boundary species values are not defined by the reaction network. Boundary species may appear as reactants or products in a reaction, but the reaction network does not alter their values; they are considered boundary conditions, hence their name. Note that either floating or boundary species may be defined via rules. In this case, such floating species may not be listed as reactants or products – the reaction network may not alter their values.

In SBML, chemical species have certain interesting behaviors, in that they may be considered either as an amount, e.g. having units of mass or particle count, or as a concentration having units of mass per volume or particle count per volume. This behavior is determined by the *substanceUnits* SBML attribute.

As SBML is intended to describe a set of reactions occurring in a CWSC, each species must reside in a compartment.

Compartments

Compartments are variables which represent the volume of a CWSC in which a set of species reside. Physically, compartments are intended to represent volume elements whereas parameters may represent any arbitrary physical value. Semantically, the only real difference between compartments and parameters is that, when a species is stated as having concentration units, the species amount is automatically divided by the compartment volume in which it resides.

When a species symbol is used in an expression such as in a rate rule or function, the *substanceUnits* attribute determines whether a species is treated as an amount or concentration. If the species is treated as a concentration, then the amount value is implicitly divided by its compartment volume when the species symbol is dereferenced. Similarly, when a species value is stored such as in an event assignment, then if the species is treated as a concentration, the value to be stored is implicitly multiplied by the compartment volume.

Function definitions

Functions in SBML are a mapping whose domain is a set of symbols in the SBML model and whose codomain is a scalar value.

As stated in the SBML reference [46], functions are intended to be used as macros. Many SBML engines use the libSBML macro expansion facility to expand function inline. Even though it is stated that functions should not reference any symbol not explicitly given as arguments, many SBML documents do reference other symbols. Even though this may not be correct according to the spec, it is a commonly seen behavior in SBML documents as a consequence of the macro expansion.

Most SBML engines just expand functions inline. This results in an interesting side effect: functions in SBML are effectively *dynamically scoped.* Dynamic scoping is used in PERL, original Lisp (before Common Lisp) and some older programming languages such as APL and SNOBOL. With this scoping behavior, functions first look for a local definition of a symbol in the given set of arguments. If it is found here, it is resolved. If not, the symbols are searched for one level up, which could be another function from which the current one was called. The search continues up the list of scoping blocks until it reaches the set of local parameters if the function was invoked from a reaction, and finally up to the global set of SBML symbols.

The compilation process for functions is discussed in detail in § 2.4.3.

Rules and Initial Assignments

A rule is a way to transform the value of one variable to another. When a rule is defined for a symbol, that symbol is treated instead as a mathematical expression which defines the value of a symbol in terms of other symbols, or defines the rate of change of a variable. Any evaluatable symbol in SBML may be defined by rules, including floating and boundary species. The caveat here is that if a species is defined by a rule, it may no longer be produced or consumed in the chemical reaction network.

SBML supports three kinds of rules: assignment, rate and algebraic. Algebraic rules define a system of implicit equations and are not supported by many SBML simulators including RoadRunner. Assignment rules operate similarly to a macro expansion in C++, or almost identically to transformation rules in Mathematica. In Mathematica, a replacement rule is defined with the arrow operator, $\rightarrow$ and applied with the replacement operator /., so the expression $x + y/.x \rightarrow 3$ evaluates to $3 + y$.

In SBML, one may define a set of SBML assignment rules (using Mathematica syntax) as

$$C \rightarrow 1 + X \tag{2.1}$$

$$A \rightarrow B + 2 \tag{2.2}$$

$$D \rightarrow A + C \tag{2.3}$$

$$P \rightarrow C + A + D. \tag{2.4}$$

If an expression requests the symbol P, it will see that the symbol is defined by a replacement rule, which in turn gets evaluated. Here the C gets replaced by $1 + X$, the A gets replaced by $B + 2$ and D gets replaced by $A + C$. The A and C are not terminals, they get replaced by $B + 2$ and $1 + X$ respectively. The only restriction is that rules cannot have a loop. So, starting with P, we have:

$$P = C + A + D \tag{2.5}$$

$$P = (1 + X) + (B + 2) + (A + C) \tag{2.6}$$

$$P = (1 + X) + (B + 2) + ((B + 2) + (1 + X)) \tag{2.7}$$

$$\tag{2.8}$$

Finally, we end up with an expression of constants and terminal symbols.

Algebraic rules in SBML also have scoping rules. The left hand side (LHS) of a rule will evaluate to a different result if the rule is evaluated inside a reaction block where there are local parameters that override the global symbol names, or at time < 0 where there may be initial assignment rules in play.

Initial assignments are just algebraic rules which are only in effect at time < 0.

Reactions

SBML was first developed to describe a reaction kinetic model. The first release of SBML, Level 1 [45] did not contain events or rate rules, thus it could only be used to describe systems in terms of a reaction kinetics model. Even though the present version of SBML can describe much more, the reaction kinetic model is still the predominant use of SBML.

A reaction specifies the transformation of a number of chemical species (reactants) into number of new species (products) at a specified rate.

In a reaction kinetics network, a network of m chemical species and n reactions can be described by the m by n stoichiometry matrix $\mathbf{N}$. $\mathbf{N}_{i,j}$ is the net number of species i produced or consumed in reaction j. The dynamics of the network are described by

$$\frac{d}{dt}\mathbf{S}(t) = \mathbf{N}(t) \cdot \nu(\mathbf{S}(t), \mathbf{p}), \tag{2.9}$$

where $\mathbf{S}$ is the vector of species concentrations, $\mathbf{p}$ is a vector of time independent parameters, t is time, and ν is the reaction rate function.

SBML allows one to define a list of reactions, and each reaction has a list of products, reactants and an arbitrary mathematical expression which specifies the reaction rate. The generation of eqn. 2.9 is discussed in § 2.4.3.

Events

Events are discontinuous state changes which are applied when a predefined condition is met. Formally, events map the model state to a new state when a predicate evaluates to true,

$$\{\mathbf{S} \rightarrow \mathbf{S}' | P(x)\} \tag{2.10}$$

See § 2.4.3

SBML Computational Model

SBML does contain numerous other elements, such as constraint or layout information, but these ancillary elements do not affect how the model is generated. The above set of SBML constructs allows the definition of an initial value problem (IVP). Formally, this IVP problem may be written as

$$\mathbf{S}(t) = \sum_{E} \int_{t_i}^{ti+1} \dot{\mathbf{S}}(\mathbf{p}_i, t) dt. \tag{2.11}$$

Here, the summation is over the set of events, and the integration proceeds from the time at which the previous event was triggered (or time 0) up to the next event trigger time (or a pre-specified halting time).

It is the role of the SBML compiler to extract and infer semantic meaning from this set of constructs in order to generate executable machine code with which this IVP is evaluated. The SBML provides sufficient information to generate a function which yields the rate of change of the entire state vector as well as the predicates and assignment rules for the set of events.

Then finally, it is the role of the SBML simulation engine to combine the generated rate of change and the event function from the compiler with an integrator, yielding $S(t)$, the state vector at time t. Extensive details of the SBML compiler are provided in § 2.4.

These SBML constructs do not explicitly define how the state vector rate and event functions are to be generated as they do in a *procedural* programming language. Rather, the compiler gathers information from all of the SBML elements to infer how to generate these functions.

In this sense, SBML can be thought of as more of a *declarative* rather than a procedural language. In a declarative language, the programmer must explicitly define the exact computation to be performed. The compiler of a declarative language must infer semantic meaning from a set of potentially disjointed information. In a procedural language, however, the onus is on the programmer to explicitly define the exact computation to be performed. The task of compiling a procedural language is typically much simpler than compiling a declarative language, as a procedural language places more of the burden for explicitly defining the computation on the programmer rather than the compiler.

Many commonly used programming languages such as C++, Java, MATLAB, Python, and so forth are considered imperative languages. In an imperative language, one writes a sequence of operations that describe in exacting detail how to perform a calculation. Although C++, Java and Python may also be considered object oriented (OO) languages, this is more a description of how the code is structured than how the computation is specified. In OO languages, data and logic are grouped together into “objects” whereas in more procedural languages such as MATLAB or C the logic is organized into procedures that operate on data. Nonetheless, these are organizational differences that do not alter the imperative nature of these languages, which is a sequence of operations that explicitly specify the computational algorithm.

On the other hand, in declarative languages like Mathematica or Prolog one defines a program by specifying the logic of a computation rather than specifying the control flow. In this sense, SBML may considered a declarative language, as only elements and relationships need to be specified.

Declarative languages pose challenges for compiler developers. In procedural languages, explicit details are provided in the source language of how the perform a computation. The compiler then has a comparatively easier job, in that it just has to determine what is the optimal way of implementing this computation on the hardware at hand. In declarative languages on the other hand, such explicit details are provided. The compiler needs to determine how to assemble all of the provided information into a description of a computation and then carry out the details of implementing this computation on the hardware.

Take for example the SBML assignment rules. These rules may be specified in any order and the order in which they are defined should have no bearing on the outcome. One may define an assignment rule in SBML as $A \to B + C$, meaning that whenever the symbol A is encountered, it is evaluates to the expression $B + C$. We may also have the rule $B \to 10$, meaning the symbol B always evaluates to the literal 10.

The fact that rules may be expressed in any order poses some challenges to software that must interpret declarative statements. In this example, it should be clear that the rule $B \to 10$ should be evaluated before $A \to B + C$. Other SBML engines such as SBW [9] introduce state variables for each intermediate rule evaluation, and a number of synchronization functions which are regularly called to synchronize the state of all the intermediate evaluations at run time. LibRoadRunner on the other hand resolves all rules at compile time so there are never any synchronization issues and there are never any redundant intermediate state variables.

2.3 Architecture

2.3.1 Design Goals

LibRoadRunner is a library designed from the start to be used in existing simulation environments. It is designed to provide a lightweight, self contained, easily embedded simulation package which provided a modern, well documented API which is natively accessible in C++ or Python, and we can easily add native bindings to many other languages supported by SWIG [8], including the upcoming support for JavaScript using either the Google V8 or Apple Web-Kit JavaScript runtimes. In addition we support a Systems Biology Workbench [9] (SBW) compatibility C API.

A number of other software packages exist which are capable of producing time series simulations of SBML models. COPASI [44] is a mature and well established program which in our testing provides better performance that the other SBML packages. It is also very feature rich. Even though COPASI may be used as a library, it appears to be more focused on being a desktop application rather than a library. The COPASI API is, compared to RoadRunner more difficult to use in that it is *significantly* more verbose, and is not well suited for interactive use. Interactive simulation and data analysis can be extremely productive and is a key feature of environments such as MATLAB, Mathematica, R or Python. In such an environment, users interactively enter commands or functions at a command prompt and these are used to perform calculations. To function well in an interactive environment, an API should not require a user to enter long code blocks, it should provide a rich set commands or functions, coupled with a range of options which can be entered rapidly in a single command line. A wide range of applications function well in interactive environments in addition to the aforementioned MATLAB or Mathematica. These include any of the UNIX command interpreters, TCL, or even the MS-DOS command prompt.

The LibRoadRunner API is designed to be used in interactive environment in addition to being embedded in existing applications. The API is designed to have the same style as the Python SciPy package. For example, to run a simple time course simulation requires over a 100 lines of code using the COPASI Python API [21]. A time course simulation can be performed using the LibRoadRunner library in Python with only two lines of code:

```
r = RoadRunner(‘‘myfile.xml’’)
s = r.simulate()
```

here `s` is numpy array in Python, or a matrix object in C++. In Python, this array may be given directly to `matplotlib` to be plotted. This example runs a time series using the default time span and automatically selects the default selections as the set of floating species. All of the simulation parameters may be specified as optional arguments, for example, to run a time course simulations with a time space from $t = 0 \rightarrow 12$, 100 data points, and outputting time series for the parameter “P1” and the concentration of species “S1”, one would run:

```
r = RoadRunner(‘‘myfile.xml’’)
s = r.simulate(0, 10, 100, sel=[’time’, ’P’, ’[S1]’], plot=True)
```

This example also uses the optional “plot” argument which automatically invokes matplotlib to plot the time series result.

The Systems Biology Simulation Core Algorithm [47], which we abbreviate as (TSBSCA) is a Java SBML simulation package which is comparable to RoadRunner as does have a cleanly designed and well structured API and has a modern and well architected internal design. The code is easy to read, understand and extend. TSBSCA does however have a confusing name, its unclear from the name of what it actually does (though that criticism can be applied equally well to the LibRoadRunner library). More fundamentally, TSBSCA is written in Java which may be ideal for other Java packages, but is not very practical to use in C++ or Python programs, in which most scientific software is developed. libSBMLSim [76] is a library written in C with an unusual coding style, however it has a limited API, only allowing loading of SBML models, and writing a result time series file. SOSLib is another library in the style of libSBMLSim, but it does not appear to be actively maintained, quoting from their web site as of March 5, 2014, “please note that we currently do not have much time to actively work on SOSLib” [72].

2.3.2 Component Based Design

The libRoadRunner is built on a component based design. All major components interact with each other via pure virtual interfaces. There is a strict separation between interfaces and concrete implementations of these interfaces. This allows us to have multiple implementations of the same interface. Most objects are created via factory objects. Here, only the factory object needs to be aware of the concrete object types. One of the primary advantages of such a design is that it allows us to have pluggable components.

Logically, we have made a strict separation between the object representing the state of the system, the ExecutableModel interact, and the object responsible for time evolution of the state, the Integrator object. In other words, the time evolution of phase space vector $\Gamma(t)$ is the result of the classical propagator [77] acting on the initial state, $\Gamma(0)$ as

$$\Gamma(t) = e^{i\mathcal{L}t}\Gamma(0). \tag{2.12}$$

Here, the phase space vector Γ is the state of the system. This is defined entirely by the source SBML document, and $e^{i\mathcal{L}t}$ is the classical propagator which advances the system in time. There are many ways of defining how a state is implemented and how a propagator acts on that state.

In libRoadRunner the state of the system is represented by the `ExecutableModel` interface, and the propagator is represented by the `Integrator` interface.

The `ExecutableModel` interface currently has two implementations: (1) a JIT compiler which directly JIT compiles source SBML documents in memory via Low-Level Virtual Machine (LLVM) (see below), and (2) a procedure (written originally by Frank Bergman and later transliterated into C++ by Totte Karlsson) which takes an SBML document and applies a series of textual transforms to generate a C language source code file. This file is then written to disk and an external C compiler is called, which in turn generates a shared library. This shared library is then loaded.

The `Integrator` interface is currently currently implemented by a number of integrators. Stochastic integration is supported by an integrator which implements the Gillespie SSA algorithm which is described in further detail in Chapter 4. Deterministic integration is supported by either simple Runge-Kutta fourth order integrator, or an very capable CVODE integrator which uses the CVODE integrator from the Sundials suite [42]. The CVODE integrator performs the temporal evolution via a number of variable-order, variable time-step methods. Non-stiff systems are evolved via the Adams-Moulton formula, using and order between 1 and 12. Stiff systems are evolved using the backward differentiation formula. We are currently developing a stochastic integrator based on the Gillespie algorithm, and are also investigating an LSODA based deterministic integrator. A variety of other methods may be investigated, such as splitting the Liouville operator $\mathcal{L}$ to develop a multi-scale propagator-based integrator. All of these approaches may be investigated with no changes to any existing RoadRunner code as the entirety of the temporal evolution system is contained completely behind the `Integrator` interface.

LibRoadRunner has a simple yet very capable object oriented public API, accessible via C++ or Python and consisting of two public interfaces and three or four configuration parameter structures. All functionality, including model loading, simulation and model variable access and analysis is available via this pair of interfaces. We expect most users to use the Python API. LibRoadRunner is available as a single self-contained Python package with extensive documentation available online at http://libroadrunner.org or interactively via standard Python doc-strings.

The LibRoadRunner API has two primary design goals, (1) to provide a rich interactive user experience when used in an interactive environment such as Python, and (2) to allow libRoadRunner to be readily used in existing applications and simulation environments. The API was developed in close collaboration with application developers [75].

Both the C++ and Python APIs interact via standard data structures. The C++ API uses only standard library data structures - i.e., `std::vector<std::string>` - or standard arrays of double precision numbers. The Python API exclusively uses standard Python lists, strings and numpy arrays for numeric data. All numpy arrays returned by libRoadRunner are thin wrappers around libRoadRunner owned memory; thus, there is no copying of any large memory blocks. For example, the matrix returned by the `simulate` method may be very large, and it would be wasteful to make copies of it. As the Python API uses standard numpy arrays, one may use libRoadRunner directly with the huge number of existing Python numeric and scientific libraries.

2.4 SBML Language Compilation

The goal of the SBML language compiler is to generate a data structure which contains all and only the required model state variables, and a series of machine executable functions which can calculate the rate of change of the state vector, allow access to the model variables and implement the set of SBML events.

To our knowledge, no other available SBML engine is capable of direct JIT compilation, and JIT compilers for declarative language are rare. Most existing SBML engines either have built-in interpreters as in the case of COPASI [44] or The systems biology simulation core algorithm [47], or macro expansion systems which apply a set of textual replacements to generate a source code file in a language for which a compiler exists such as C or Java as in the case of the Systems Biology Workbench [9]. There was however a system developed by Ackermann et. all. [1] which was capable of generating CUDA code from SBML and executing it on an nVidia GPU. This system appears to be very limited as it appears to only handle systems which consist of only rate equations (no events or rules), and there is no available source code or binary, and it appears to be only capable of accepting SBML rate rules, and is limited to eight state variables.

A Just In Time (JIT) compiler is a routine which which takes the source code description of a computational process and performs in-memory translation into machine executable routines immediately before they are required for use. In contrast, an interpreter either directly executes the source code program (though it frequently generates a more efficient in-memory representation). In an interpreter, each time a statement is run, a considerable amount of logic must take place to determine the exact form and intended operation of the statement. Interpreters are typically 50-100 times slower [50] than native executable code.

Systems such as the SBW [9] convert a SBML file into a C# file through a series of textual transforms and macro expansions. It then uses an external compiler to generate a shared library which then finally contains a set of native machine executable functions. Such macro systems are not considered a true compiler, rather it is more of "a cousin of a compiler", the preprocessor, (see sec 1.4, [2]) which produces input to compilers. Preprocessors may be capable of limited computation such as arithmetic or logic operations, but their primary purpose is take a source document, perform some transforms or expansions and prepare input for compiler. Some other examples of preprocessors are the C preprocessor, the Qt MOC (meta object compiler) or the current crop of languages built on top of JavaScript such as Coffee Script. Source code translation or pre-processor systems are typically slower than a JIT compiler for a number of reasons such as they have to access the file system which is an order of magnitude slower then in-memory operations and they are calling a general purpose compiler which accepts language considerably more complex than SBML and as such, can take considerable time to run.

At its simplest level, SBML is a language for describing a system of ordinary differential equation (ODEs) in the form of

$$\frac{d}{dt}\mathbf{S}(t) = \begin{bmatrix} \dot{\mathbf{S}}_f \\ \dot{\mathbf{S}}_r \end{bmatrix} = \begin{bmatrix} \mathbf{N}(t) \cdot \nu(\mathbf{S}(t), \mathbf{p}) \\ f(\mathbf{S}(t), \mathbf{p}) \end{bmatrix}, \tag{2.13}$$

where $\mathbf{S}$ is the total state vector of the system from eqn. 2.11. The state vector is partitioned into two vectors, $\mathbf{S}_f$ is the vector of independent floating species which participate in a reaction network, and $\mathbf{S}_r$ is a vector of variables which are defined by rate rules. The vector $\mathbf{p}$ consists of time independent parameters.

The chemical reaction network of m chemical species and n reactions can be described by the $m \times n$, potentially time-dependent stoichiometry matrix $\mathbf{N}(t)$. Each stoichiometric element, $\mathbf{N}_{i,j}$ is the net number of species i produced or consumed in reaction j, and $\nu(\mathbf{S}(t), \mathbf{p})$ is the function yielding the length n vector of reaction velocities.

The second part of the state vector, $\mathbf{S}_r$ is a set of variables which form a system of conventional ODEs, i.e. each element in this vector is defined by an SBML rate rule. Any SBML element including floating species may be defined by rate rules. When a floating species is defined by a rate rule, we no longer consider it an independent floating species as it is not defined by a set of reactions. In this sense, floating species defined by rate rules behave semantically more similarly to boundary species rather than floating species.

In order to generate the full time evolution of the state vector (2.11), integration needs to be partitioned into a finite set of discrete time intervals which are determined by the set of events. The state vector rate and event functions in turn are native machine executable functions which are generated just in time (JIT) by the SBML compiler. The following section is very high level overview of compiler design. Those already familiar or not interested in compiler design may skip ahead to section § 2.4.3.

2.4.1 Overview

The process of loading, compiling and simulating an SBML model is partitioned between the following classes shown in fig. 2.1. The `RoadRunner` is the top level class which provides a facade and manages the interaction of other child classes. The `ModelGenerator` is where all the JIT compilation takes place. This class generates an `ExecutableModel` class which contains all of the JITed code as well as a buffer which holds the state vector, initial conditions and other model meta data. The `Integrator` class queries the `ExecutableModel` for the state vector rate and performs the time evolution of the system.

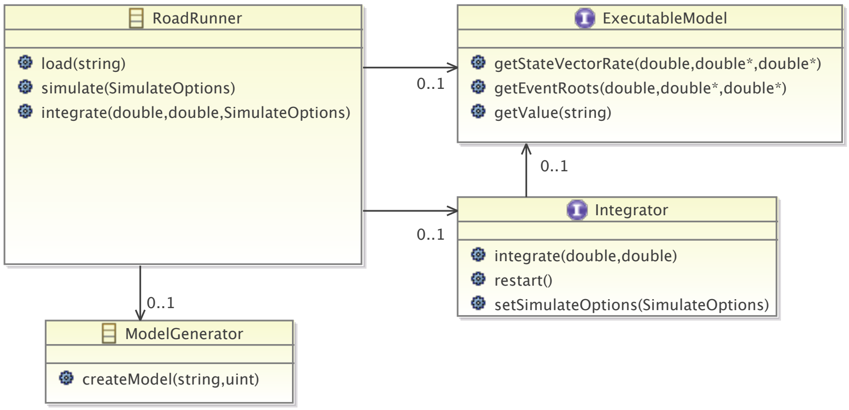


Figure 2.1: A simplified UML diagram of the key objects involved in SBML JIT compilation and simulation.

2.4.2 Compiler Design Overview and LLVM Details

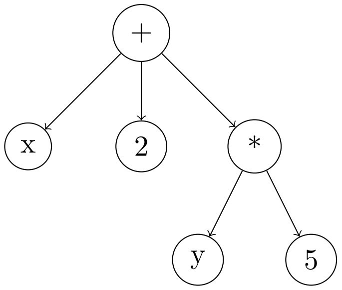
A compiler is a computational procedure for translating a source document, typically in human readable text into executable machine code. The phases most compilers execute are: (1) lexical (2) syntactic, (3) semantic (4) intermediate code generation, (5), code optimizer, and (6) native code generator. Phases 1 through 4 are the analysis phase. Here, the source code is separated into parts and then arranged into a meaningful structure (or grammar of the language). Stages 5 through 6 are the synthesis phases. It is here that the actual executable machine code is generated.

The initial and final stages of compilation are fairly well defined, generalizable and many excellent libraries exist for performing these tasks, so they will not be covered in any detail here. The medial stage, semantic analysis, is however very specialized to the programming language which is being compiled. This stage will be covered in detail in the following sections.

In the lexical and syntactic analysis phases, a sequence of characters in the source program are categorized and grouped into sequences called lexemes. For each lexeme, an abstract symbol called a token is produced. Lexical analysis is concerned with syntax and other things. The result of the lexical analysis phase is typically a parse tree. The semantic analysis is concerned with the meaning of the source program. This is where the meaning of the source program is determined and a form suitable for machine???

There are many well defined packages for dealing with the lexical and syntactic analysis phases. If the source file is in textual source format, it one may specify the BNF grammar and use a program like Yacc, Bison [23] or ANTLR [60] to automate the generation of a syntax analyzer. As SBML is XML, it can be thought of as a textual encoding of a parse tree. Therefore, the syntactic rules are much smaller than a typical programming language. The the libSBML [13] library is used to perform the entire lexical analysis phase. The libSBML library provides a DOM (document object model) data structure which is in effect, a parse tree and is comparable to the parse tree data structures provided by parser generators such as ANTLR. The result of the syntactic analysis phase is typically an *abstract syntax tree* or AST.

The AST is a data structure which contains the essential semantic information which is required by the later stages if a compiler (semantic analysis, code generation) from a source program. An AST is abstract in the sense that it does not contain *all* of the information in the source program, all of the semantically irrelevant information has been removes such as whitespace, comments, parenthesis, etc... Each node in the AST represents an essential construct such as operators, symbols, literals, function application, etc... An example of source language string and the corresponding AST are depicted in Table 2.1.

Infix	MathML	AST
`x + 2 + (y * 5)`		

```
<math>
  <apply>
    <plus/>
    <ci>x</ci>
    <cn>2<cn>
    <apply>
      <times/>
      <ci>y</ci>
      <cn>5</cn>
    </apply>
  </apply>
</math>
```

Table 2.1: A mathematical expression expressed as infix notation, MathML notation and as an AST.

In the final stages of the compilation process the intermediate representation is optimized and finally turned into executable machine code, the LLVM library is used to perform this task. LLVM can produce executable code for a variety of architectures; of particular interest is GPU format.

LLVM Intermediate Code Representation

The LLVM intermediate representation (IR) is an *single static assignment* or SSA language. This states that each variable may be assigned exactly once, there can be no storing of any value in an existing variable. Any new calculated value must be stored in a new variable. In contrast, Java byte-code Microsoft IR used in the .net platform are both stack based intermediate languages.

2.4.3 Implementation of SBML Language features

In a procedural language, each symbol almost invariably corresponds to a single particular location in memory, be it a variable or a function. This mapping of language symbols to memory locations is typically handled with data structure known as a *symbol table*. One may use multiple symbol tables, such as when function with local variables. Here, when the compiler generates code for a function, a new symbol table is creates which holds the set of local variables. The compiler first looks in this local set to resolve a symbol, then in the global table. This approach can be extended further, such as in Pascal which supports nested scoping. Here, function can be defined inside other functions, and symbol resolution proceeds from local to a chain of parent functions then finally to the global scope.

The conventional symbol table approach becomes problematic when dealing with declarative languages. In SBML, symbols are often defined by rules and symbol resolution is context dependent. A symbol may resolve to different values if it is evaluated before or after $time = 0$, and if it is used inside a reaction with local parameters. Completely different sets of rules may be in play depending on the model time of evaluation.

One approach would have been to simply allocate storage space for *all* symbols. This however would be wasteful in terms of memory usage and would result in significantly more complex and error prone code at run time. The would require numerous other functions to to evaluate rules and store their results before they are used.

The approach taken here is an extension of the symbol table which we refer to as a *symbol forest*. This is a hash table which maps symbol names to ASTs. In effect, this contains all the un-evaluated rules. These rules are resolved and evaluated when they are called. Thus, this is a form of lazy evaluation. This approach allows us to only allocate storage for terminal nodes and never requires any updating of intermediate variables. Everything is evaluated in-line. One may be concerned that this approach might incur redundant evaluations, but the LLVM optimization passes eliminate redundant operations.

Only terminal symbols (symbols not defined by assignment rules), reaction rates, stoichiometric coefficient and initial conditions are allocated storage. All variables are stored in a single contiguous memory block. This approach allows us to compute the offset of each variable at JIT compile time and results in fewer memory accesses. If each type of variable were stored in a separate dynamically allocated array, it would have incurred an additional an additional memory access for each variable reference, and the resulting code would have more difficult to optimize for the LLVM optimizer passes.

Terminal symbols are accessed via a pair of pure virtual interfaces, `LoadSymbolResolver`, and `StoreSymbolResolver`. These pair of interfaces are our equivalent to the traditional symbol table, these map a symbol to either a load or store instruction. Each of these resolver interfaces have a number of concrete implementations to accommodate the various SBML scoping rules. Resolvers can also be chained, e.g. A `FunctionResolver` resolves symbols to function arguments, and this is chained to a parent resolver which may be from a calling function, or somewhere up the scoping stack, until finally all symbols must be resolved at the terminal resolvers such as the `ModelDataLoadSymbolResolver` which maps symbols to locations in the state vector.

Storing the entire model (and consequently the entire state vector) as a single contiguous memory block also allows significant performance optimizations when interacting with the integrator. The integrator calls the `getStateVectorRate` function. This function calculates the rate of change of the state vector as a function of the state vector. This function does not need to copy any memory, it only needs to swap out a pair of pointers specifying the base address of the state vector and state vector rate. The memory layout is exactly the same as what the integrator expects: a single contiguous array of state vector variables.

Species, Compartments, and Parameters

In SBML, variables may be species, compartments or parameters. Compartments and parameters which are terminal symbols are treated as conventional variables in a procedural language: they are allocated a region of memory and may be written or read from this location. Species have a different semantic meaning. Species may be treated as either a concentration (amount/volume), or an an amount. Our implementation *only* stores amounts, concentration are never stored in memory. Whenever a species symbol is evaluated, and it is determined that this is a concentration type, the generated code automatically divides the amount by the compartment volume in-line. Similarly, whenever a species is to be stored, the generated code automatically converts any concentration store into an amount store operation.

Such a lazy evaluation approach ensures that there are never any synchronization issues with variable compartment volumes and concentrations.

Rules and Initial Assignments

Assignment rules are expanded incline during code generation process walking of the AST. When a symbol is dereferenced, it is first looked up in the symbol forest. If this symbol turns out to be a terminal symbol (not defined by an assignment rule), then that symbol ends up as being mapped to particular location in the state vector and load or store instruction is emitted. If however this symbol has an assignment rule, then the symbol forest returns the root of the assignment rule, and the code generation continues here.

If a symbol is defined by an assignment rule, then it is never allocated any space in the state vector. It is an error to attempt to set the value of symbol specified by an assignment rule.

Initial assignment are handled almost identically, the only difference being that the terminal symbols map to different memory locations. All initial condition values are actually stored in a different location that the state vector, as a call to `reset` sets the state vector variables to the values specified by the initial conditions.

Functions

Functions in SBML have peculiar behavior. In the SBML specification, it is stated that functions are intended to act as “macro expansions”. This has the implication that any symbol referenced in the function body must be resolved to variable one level up the call stack. Therefore, SBML functions are *dynamically scoped.* Note, even though the specification states that the only symbols available in a function are those that are explicitly passed in, frequently one encounters SBML with symbol references which are not resolved to function scope. Many other SBML engines gladly accept these functions, as they simply, as the specification states, expand the function as a macro.

Reactions

Simulation of reaction kinetics networks is one of the primary uses of SBML. In order to calculate the time course of the reaction network portion of the state vector, (2.13), we must first specify how the stoichiometry matrix is stored, generate a function which calculates the reaction rate vector, and perform the matrix vector product.

The SBML JIT compiler generates a `getStateVectorRates` function which is called by the integrator. The is the most frequently called routine in all of libRoadRunner so performance in critical. This function receives a pointer to the current state vector, the address of all SBML symbols is known at compile time and is a fixed offset from the base address of the state vector, and the resulting state vector rate values are written directly into a memory block owned by the integrator. This, there is no copying of any memory during this call.

The SBML JIT compiler also generates a `getReactionRates` function which implements the ν function in (2.13). The resulting vector is stored in a memory block.

As the stoichiometry matrix $\mathbf{N}$ is typically extremely sparse, it is stored in compressed sparse row (CSR) form. Thus the matrix vector can be calculated in $\mathcal{O}(n_z \times n)$ time, where n_z is the number of non-zero elements and n is the number of reactions instead of $\mathcal{O}(m \times n)$, where m is the number of chemical species.

During the course of the matrix-vector product, each stoichiometric coefficient must be accesses n times, thus it is read typically much more often that it needs to be written, even in the case of time dependent stoichiometries. When any SBML expression (rate rules or events) change the value of time dependent stoichiometries, the symbol forest maps this SBML symbol to the appropriate location in the CSR matrix. The CSR matrix vector product is highly optimized as in the course of our testing, nearly 30 % of the time spent calculating the state vector rate is spent here. The result of the CSR matrix vector product is calculated directly into the output buffer which is owned by the integrator.

Events

An SBML model may contain a collection of events. An event in SBML is an object which consists of the following: 1) a predicate whose variables may be any symbol in the model, 2) a set of instantaneous, discontinuous model state changes which are applied at some time (may be zero) after the predicate evaluates to true, 3) a delay function which calculates the time span between event trigger time and event application time, 4) a priority function which calculates the event priority, and 5) a pair of static boolean attributes:

useValuesFromTriggerTime and *persistent*. All event functions are functions of the model state. The useValuesFromTriggerTime attribute indicates that the assignment rules should be evaluated at the moment the event is triggered. Thus, if this attribute is set in our implementation, the event object also contains a data block where the results of the assignment rules are stored whilst the event is in the triggered state.

An event may be in one of three states: 1) Inactive, 2) Triggered, or 3) Assigning, as depicted in Figure 2.2. Events transition from the inactive to the triggered state at the moment the predicate evaluates to true. This is referred to as triggering the event. Events may also transition from triggered to inactive if their predicate evaluates to false and they are not persistent. Events transition from triggered to assigning once their application time (triggered time + delay time) elapses and their predicate is still true. Events in the assigning state perform the model state changes and transition back to the inactive state.

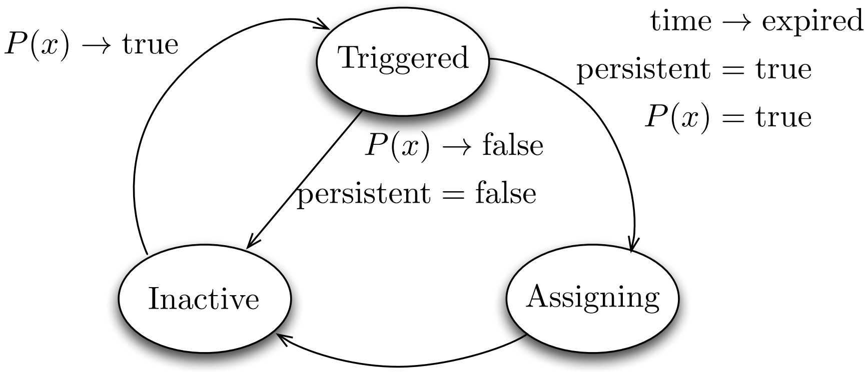


Figure 2.2: A state diagram of the events system

When the SBML model is loaded, the predicates, assignment rules, priority and delay functions for each event are JIT compiled and a function pointer to each one of these is retained.

During the integration process, the integrator asks if any events have been triggered. If any event predicates change state, the integrator performs a root finding process to determine the exact moment an event was triggered. This is required as the integrator may be taking large time steps and all that is known is that the event predicate was false at the beginning of the time step and transitioned to true at the end of the time step. An event is inserted into a priority queue at the moment it is triggered. The priority queue is sorted first on the time span between the present time and the time at which the event is to be assigned, second on the event's priority.

If any event predicates change state, any expired events are removed from the priority queue, and any *ripe* events are applied and removed from the queue. Ripe events are those which are ready to be assigned – whose application time has passed and are still active (either their persistent attribute is set, or their predicate is true). The assigning of an event by definition incurs a model state change. This state change may trigger new events, and may cause existing events to expire. Therefore, the application of events must be performed iteratively. On each iteration, expired events are first removed. As the priority queue is sorted first on time until the event is to be assigned, only the top most events will have the time to assignment as zero, therefore, only the top most events may be ripe. The set of ripe events with equal priority and assignment time are then removed from the queue, and applied in random order. The assignment of ripe events in random order is needed to comply with the SBML specification. All events are then scanned and any events that are triggered as a result of the previous event application are inserted into the queue. The iteration continues until there are no more ripe events. Once all ripe events have been assigned, the integrator can continue normally until the next event is triggered or the total simulation time span is complete.

SBML Events Performance Aspects

The processing of SBML events is currently one of the most time consuming calculations in RoadRunner. Most tests in the SBML test suite take on average approximately 0.01 seconds (on a 2.66 GHz Mac Pro), whereas the longest test, number 966 takes 1.9 seconds. A performance breakdown is in Table 2.2.

The CVODE root-finding system, which uses the Illinois algorithm, a modified secant method [24, 41] is currently used to find the event trigger zero crossings. This algorithm is well suited for smooth functions, however it can converge very slowly for functions with discontinuous zero crossings, as is the case with our event root function.

Whilst the more sophisticated root finding methods such as secant or false-position

method	% time	% self time
integrate	96	13
cvodeRootFind	70	53
rootFindCallback	17	17
applyEvents	10.5	10.5
getEventRoots	8	8
read and JIT compile SBML	2.3	2.3
integrateCallback	0.6	0.6

Table 2.2: Percent of total time and self time spent in various methods. The total time includes all the time spent in child methods. Most of these are aggregate values in that the CVODE integrate method call numerous other methods such as linear algebra routines. All of these are are considered contributors to the self time.

typically perform better than the bisection method in general, their performance may be worse in certain cases cases, e.g., when the slope of the function changes rapidly (or in the extreme case, is discontinuous) around the root. Furthermore, these more sophisticated methods are more complex to implement than the bisection method and considerable time may be spent here, as evidenced in Table 2.2. Therefore, in future versions, a root finding system based on the bisection method which should yield significant performance increases.

Code Generation

Accessor Functions

As mentioned earlier, RoadRunner only stores independent state variables. However, *all* SBML symbols are accessible trough the public API. All symbols are access through a set of generated accessor functions, and all rules are evaluated when the functions are generated.

2.5 Results

2.5.1 Performance

To demonstrate the capabilities of libRoadRunner we compared it to three simulator libraries: libSBMLSim, COPASI and SBSCL using a variety of standard and contrived models.

In our testing, we have found performance to be highly compiler and machine specific. The Linux binaries that we provide are purposely compiled on a very conservative platform, (RHEL5, GCC 4.4, with processor optimizations set to Intel Core2 level). This binary however performs 30% when running the SBML test suite than a binary compiled with GCC 4.8.1 on an 3.0 GHz Intel Core2 Quad Ubuntu 13.10 machine even when set to the same optimization level when running the the same hardware platform. At this point, it is unclear if this performance discrepancy is due to compiler differences, or library differences (the binary compiled on the older machine will still reference older symbols even when running on a newer hardware platform). Test suite performance is highly I.O. bound: when run from a clean boot, with no cached files, the test SBML test suite completes in $\approx 30-40$ seconds, whereas subsequent tests (after the operating system has cached the input files) complete in ≈ 11.5 seconds.

We purposely did not compare total run times of the SBML test suite for the different simulator libraries as all of them operate on a different set of tests (COPASI and LibRoadRunner operate each on a different subset and LibSBMLSim and SBSCL operate on all tests). Also note the tests performed here were done with binary distributions of LibSBMLSim and SBSCL, we are unaware of what compiler was used to build them. LibRoadRunner does however pass the SBML test suite tests without delay equations, algebraic rules, and completes all 1016 at current count of these tests including result validation in ≈ 11.5 seconds on a 3.0 GHz Intel Core2 Quad Ubuntu 13.10 machine, using a single core.

Most of the tests performed here are far more a test of the integrator rather than the SBML library. In LibRoadRunner only $\approx 1-4\%$ of the total time is actually spent in the JIT compiled code evaluating the state vector rate of change. When running very short simulations such as the SBML test suite, $\approx 50-60\%$ is typically spent in reading, parsing and JIT compiling the model, reading and parsing are performed by LibSBML and the JIT compilation is performed by our JIT compiler. Finally, depending on the length of the simulations, $\approx 30-60\%$ of the total time is typically spent in CVODE, this fraction of total time is much larger when using the CVODE stiff (implicit backwards difference) solver. For very long simulations, the fraction of time spent in CVODE will tend $\approx 90\%$. In such cases, the remaining fraction is consumed by selecting output variables, state vector rate evaluations and various overhead. We do not have a performance breakdown of the other libraries as we are un-familiar with their internal code structure

LibRoadRunner uses the CVODE solver (version 2.5.0 at time of publication) for integrating deterministic systems. We support both the Adams-Moulton method for non-stiff systems, and the Backward Differentiation method for stiff systems. In both cases, an implicit system of the form $0 = y^n - (\cdots)f(t_n, y^n) - a_n$ must be solved. Non-stiff systems are solved with functional iteration, whereas stiff systems are solved with Newton iteration. LibRoadRunner offers a flag which can be set before an integration time step is performed to switch between stiff and non-stiff systems (or even different integrators). COPASI on the other hand uses the LSODA [61] solver, which automatically switches between non-stiff and stiff methods dynamically depending depending on the system behavior. All tests were performed using both stiff and non-stiff LibRoadRunner solvers.

All tests were prepared in SBML test suite format, with each SBML model having an accompanying settings text description file. These are available from our GIT repository. LibRoadRunner, COPASI and SBSCL natively support the SBML test suite format.

All test times were recorded using the `‘‘real’’` value from UNIX time command. In order to compare against COPASI, we used the `‘‘rr-sbml-benchmark’’` program which was created to have the identical command line argument signature to COPASI's `sbml-testsuite` program. The SBSCL times were measured by timing the command `‘‘java -cp \ $SBSCL/SimulationCoreLibrary_v1.3_incl-libs_src.jar \ org.simulator.SBMLTestSuiteRunner $SIMDIR $NTEST $NTEST’’`, where 64 bit Java version 1.6.0_65 was used. A shell script was written which parses the SBML settings file and prepares the appropriate command line arguments to the LibSBMLSim `‘‘simulateSBML’’` program, which was used for the LibSBMLSim timings. LibSBMLSim was run using the Runge Kutta Fehlberg (RKF) integrator, which is what appears to be used in the LibSBMLSim `‘‘runall.sh’’` script to run the SBML test suite. This is explicit finite difference scheme capable of adaptive time stepping. According the SBSCL Java Documentation, SBSCL uses a Rosenbrock method based solver. The Rosenbrock method is an adaptive, semi-implicit, multi-stage method. In a sense, it is similar to the RKF method used in LibSBMLSim in that it is a generalization of the Runge-Kutta method.

The first block of tests consist of a model prepared from N copies of the Brusselator model, and a the piecewise model consists of the $\sin(t)$ function implemented as a 63 element piecewise MathML function. The piecewise models have parameter(s) defined by a rate rule, and the rate of these parameters is the piecewise sin(t) function. The piecewise test was specifically written to exercise the model state vector rate evaluations in the generated / interpreted code. The Brusselator is a fairly simple, non-stiff oscillating system which is discussed further in § 2.5.4. Each instance of the Brusselator model consists of four state variables, e.g. the 500 model in Table 2.3 consists of $20,000$ state variables. This system was written to test how well the SBML libraries cope with large systems. The timings are the total wall time (`‘‘real’’` value from the Unix `‘‘time’’` command) to complete each process. Tests were run on a 2.6 GHz Mac Pro, OS X 10.6, and used COPASI v. 4.9.43, libSBMLSim v. 1.1 and SBSCL v. 1.3. Note, we experienced instability of libSBMLSim runs with >100 copies of the Brusselator system or complex SBML.

# Brusselators	libRoadRunner - std	libRoadRunner - stiff	COPASI	libSBMLSim	SBSCL	# Rate Rules	libRoadRunner - std	libRoadRunner - stiff	COPASI	libSBMLSim	SBSCL
50	0.5	0.8	0.9	12.2	13.0	1	1.60	1.84	12.3	N/A	2.4
100	0.9	2.3	3.8	50.5	46.1	2	2.14	2.61	24.1	N/A	13:20
150	1.1	4.4	7.2	N/A	1:52	3	2.71	3.68	35.2	N/A	39:52
200	1.9	7.2	13.2	N/A	3:51	4	3.39	4.83	46.5	N/A	1:32:32
250	2.6	11.1	21.8	N/A	8:37	5	4.20	6.48	58.4	N/A	3:04:21
300	3.3	15.6	30.1	N/A	14:09						
350	3.9	21.5	46.3	N/A	21:11						
400	4.7	28.6	55.7	N/A	33:35						
450	5.6	36.3	1:14	N/A	48:12						
500	6.6	44.5	1:35	N/A	1:14:21						

Table 2.3: All times are in seconds. The first set of tests consisted of N copies of the Brusselator system in a single model. The second set was a sin function implemented as a 63 element piecewise SBML functions combined with N parameters defined by rate rules integrating the sin function. Test models are available at libRoadRunner.org.

The first set of results indicate that in the case of non-stiff systems, the RoadRunner non-stiffs solver clearly scales linearly with system size and the stiff solver scales quadratically. The Brusselator is a non-stiff periodic systems which causes the CVODE non-stiff solver to operate in a fairly fixed time step regime, and here, this solver scales directly with system size. Also, the generated LLVM stores the stoichiometric coefficients in compressed sparse row (CSR) format, the number of reactions is directly proportional to the system size, and the CSR matrix-vector product is proportional to the number of non-zeros in the stoichiometry matrix by the reaction rate vector size. Thus, the both the ODE solver, and the state vector rate calculations scale directly with system size.

The stiff solver on the other hand scales quadratically with system size. Here, the state vector rate calculation is still proportional to system size, however the backward difference formula needs to solve a linear system, and this operation is scales quadratically with system size.

The right hand block of Table 2.3 shows the results of the piecewise test. This model specifically stresses the state vector rate evaluation code, as the system dynamics are purposely designed to be non-stiff and to have simple behavior. As the RoadRunner state vector rate is calculated with JIT compiled native x86-64 code, this function could have at most 63 BNE (branch on negative) instructions. The other SBML engines all use interpreters which results in significantly longer run time.

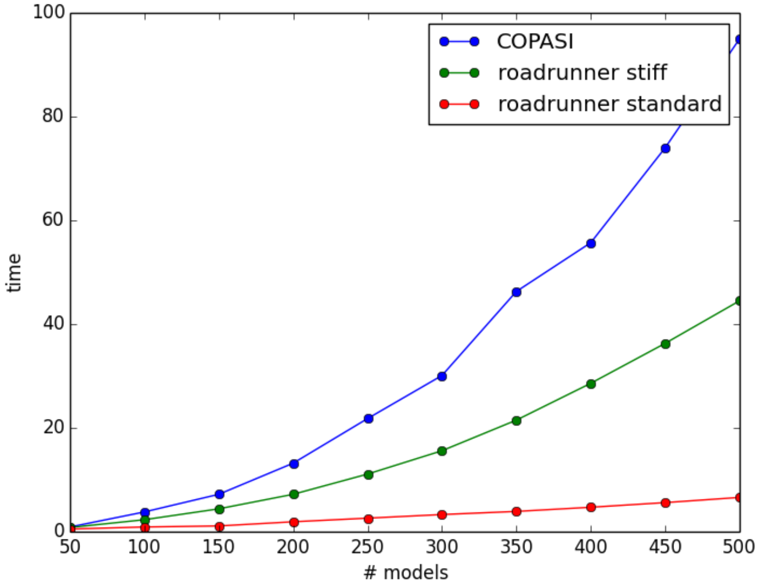


Figure 2.3: Run time performace relative to system size for the multiple Brusselator system.

The second set of tests were the models used in the original SOSLib paper [56]. The upper block in Table 2.4 was evaluated by us using the above testing procedure. The lower block is reproduced from the SOSLib web site located at [73]. This second set of data is strictly here for the sake of completeness and should not be compared directly to the first block, as we have no information on the test procedures or the type of hardware / operating system, although we suspect that the original SOSLib tests were most likely performed on a MS Windows computer as they test Jarnac which is a Windows only program.

The models uses in these series of tests had relatively few numbers of state variables, were run for short duration, but were stiff. Models $9-33$ correspond to the same numbered models from BioModels database. The repressilator was downloaded directly from the SOSLib web site at [71]. Models 9 through 22 have wildly varying initial transients, but rapidly reach a steady state value.

model	9	14	22	33	repressilator
simulation time	150	300	2000	60	10e4
# state variables	22	86	28	10	6
absolute	1e-15	1.0e-4	1.0e-4	1.0e-4	1.0e-4
relative	1e-9	1.0e-9	1.0e-9	1.0e-9	1.0e-9
dynamics	T / S	T / S	T / S	oscil	oscil / stiff
libRoadRunner - stiff	95	510	230	125	1,040
libRoadRunner - std	760	920	235	180	3,320
COPASI	200	1,980	510	250	1,600
SBSCL	1,700	6,950	25,300	2,200	98,000
2005 SOSLib Data *Not comparable to above data					
absolute	1.0e-4	1.0e-4	1.0e-4	1.0e-4	1.0e-14
relative	1.0e-9	1.0e-9	1.0e-9	1.0e-9	1.0e-9
Dizzy 1.11.1 ODEtoJava-dopr54-adaptive	15,499	12,711	2,634	19,350	6,369
Jarnac 2.16n	344	14,531	1,157	5,843	4,516
SBMLToolbox MatlabR14SP3ode15s	188	920	302	5,554	6,681
COPASI 4.0 Build 15	156	4,062	109	1,437	500
SOSlib 1.6.0pre from CVS, Nov. 17th 2005	234	515	171	562	1,062

Table 2.4: A listing of performance times from the BioModels models that were originally used to test SOSLib. All times are in milliseconds.

This second sets of tests complete in very short amount of time. The short duration of these tests would tend to give an advantage to interpreter based SBML engines and the time for setting up an interpreter is typically faster than JIT compiling. The very short nature of these tests would also tend to penalize Java based engines as there is a much longer program load time for Java vs native compiled programs.

Because of the stiff nature of these models, there is a significant advantage in using the stiff implicit solver. Here, in order to achieve any numerical stability, the non-stiff solvers end up using an extremely small time step which results in very long run times. The implicit stiff solvers are unconditionally stable and can use larger time steps (at the cost of solving a more complex problem), furthermore, the stiff solvers can better adapt time step size than the non-stiff counterparts.

In summary, SBML run time performance is highly operating system, model and integrator specific. There is no single choice of integrator that is universally better. In LibRoadRunner we have provided a default configuration which performs well with the SBML test suite, however based on the type of model used, this may be inappropriate. Therefore, our API makes it very simple to change integration behavior. All of the integration tuning parameters, such as stiff vs non-stiff, error tolerance, internal integrator parameters such as internal time step details, etc are all specifiable as either optional keyword arguments to the python “RoadRunner.simulate“ method, or the default values may be specified in a configuration file.

2.5.2 Analysis features

The LibRoadRunner library supports a wide range of analysis functions, these are described briefly here.

The stoichiometric matrix defines the biochemical network, the library provides a suite of functions to determine various subspaces of this matrix, these are implemented internally by the libStruct library [66].

The dynamics of a biochemical network is described by the system equation

$$\frac{d}{dt}\mathbf{s}(t) = \mathbf{N}\mathbf{v}(\mathbf{s}(t), \mathbf{p}, t), \tag{2.14}$$

where $\mathbf{s}$ is the vector of species concentrations, $\mathbf{p}$ is a vector of time independent parameters, and t is time. The steady state is the solution to the network equations when all the rates of change are zero. That is the concentrations of the floating species, $\mathbf{s}$ that satisfy:

$$\mathbf{N}\mathbf{v}(\mathbf{s}(t), \mathbf{p}, t) = 0 \tag{2.15}$$

The library is designed to support multiple steady state solvers. Currently NLEQ steady solver from [58] is used, however a number of additional solvers such as KINSOL from the Sundials suite [42] is planned. The steady state of the system is calculated with a single call to the `steadyState()` function.

Metabolic control analysis is the study of how sensitive the system is to perturbations in parameters and how those perturbations propagate through the network. Two kinds of sensitivity are defined, system and local. The local sensitivities are described by the elasticities. These are defined as follows:

$$\varepsilon_S^v = \frac{\partial v}{\partial S}\frac{S}{v} = \frac{\partial \ln v}{\partial \ln S}$$

Given a reaction rate v_i, the elasticity describes how a given effector of the reaction step affects the reaction rate. Because the definition is in terms of partial derivatives, any effector that is perturbed assumes that all other potential effectors are unchanged.

The system sensitivities are described by the control and response coefficients. These come in two forms, flux and concentration. The flux control coefficients measures how sensitive a given flux is to a perturbation in the local rate of a reaction step. Often the local rate is perturbed by changing the enzyme concentration at the step. In this situation the flux control coefficient with respect to enzyme E_i,

$$C_{E_i}^J = \frac{dJ}{dE_i}\frac{E_1}{J} = \frac{d\ln J}{d\ln E_i}.$$

Likewise the concentration control coefficient is as

$$C_{E_i}^S = \frac{dS}{dE_i}\frac{E_1}{S} = \frac{d\ln S}{d\ln E_i},$$

where S is a given species. The response coefficients measure the sensitivity of a flux or species concentration to a perturbation in some external effector. These are defined as,

$$R^J_X = \frac{dJ}{dX}\frac{X}{J} = \frac{d\ln J}{d\ln X}$$
$$R^S_X = \frac{dJ}{dX}\frac{X}{S} = \frac{d\ln S}{d\ln X}.$$

where X is the external effector. All of the analysis features were originally written by Herbert Sauro and are provided as a set of functions in the LibRoadRunner library and full documentation is available on the library website at [28].

2.5.3 Conserved Quantities

Overview

The term moiety originated from medieval French *moitié*, meaning a parcel or portion of a larger system that has been divided. Conserved moieties in chemistry are aggregate groupings of chemical species that are conserved in a system, regardless of the individual reaction rates. Consider the following system:

$$\begin{aligned} A &\rightarrow B \\ B &\rightarrow C \\ C &\rightarrow A \end{aligned}$$

Regardless of the rates of reactions the quantity $A + B + C = constant$ is conserved for all time. Such conserved quantities are considered structural properties of the chemical network as they are defined by the structure of the network, and not by the dynamics of the rate equations. In systems biology, such a conserved quantity is typically referred to as a "conserved moiety." A common example of a conserved moiety is the conservation of the adenine nucleoside moiety, i.e., the total amount of adenine in its various forms, AMP, ADP, ATP, is conserved throughout the time evolution of the system. Finding and analyzing conserved moieties can yield insights into the structure and function of a biological network. Conserved moieties represent dependencies that can be removed to reduce a system's dimensionality, or number of dynamic variables. In the previous simple network, it is only necessary to determine two out of three state variables, i.e., one only need determine either $A(t)$ and $B(t)$, then $C(t)$ is known from a linear combination of A and B. Similarly, if $B(t)$ and $C(t)$ are known, then $A(t)$ can be determined.

Moiety Conservation Analysis

A network of m chemical species and n reactions can be described by the $m \times n$ stoichiometry matrix $\mathbf{N}$, $\mathbf{N}_{i,j}$ is the net number of species i produced or consumed in the reaction

$$\frac{d}{dt}\mathbf{S}(t) = \mathbf{N}\mathbf{v}(\mathbf{S}(t), \mathbf{p}, t), \tag{2.16}$$

where $\mathbf{S}(t) = [S_1(t), S_2(t), \cdots, S_n(t)]^T$ is the time dependent vector of independent chemical species concentrations, N is the time invariant stoichiometry matrix,
$\nu(t) = [\nu_1(t), \nu_2(t), \cdots, \nu_n(t)]^T$ is the time dependent vector of reaction velocities, $\mathbf{p}$ is a vector of time independent parameters, and t is time.

Each structural conservation, or interchangeably, *conserved sum* (e.g., conserved moiety) in the network corresponds to a linearly dependent row in the stoichiometry matrix $\mathbf{N}$. If there are conserved sums, then the row rank, r of $\mathbf{N}$ is $< m$, and the stoichiometry matrix $\mathbf{N}$ may first be re-ordered such that the first r are linearly independent, and the remaining $m - r$ rows are linear combinations of the first r rows,

$$\mathbf{N} = \begin{bmatrix} \mathbf{N}_r \\ \mathbf{N}_0 \end{bmatrix}. \tag{2.17}$$

The *reduced stoichiometry matrix*, $\mathbf{N_r}$ consists of the first r (independent) rows of $\mathbf{N}$, and rows of the dependent stoichiometry matrix $\mathbf{N}_0$ are formed by linear combinations of the independent rows of $\mathbf{N}_r$. $\mathbf{N}$ may be expressed as a product of the $m \times r$ *link matrix* $\mathbf{L}$ and the $r \times n$ $\mathbf{N}_r$ matrix:

$$\mathbf{N} = \mathbf{L}\mathbf{N_r}.$$

The link matrix $\mathbf{L}$ has the form

$$\mathbf{L} = \begin{bmatrix} \mathbf{I}_r \\ \mathbf{L}_0 \end{bmatrix},$$

where $\mathbf{I}_r$ is the $r \times r$ identity matrix and $\mathbf{L}_0$ is a $(m-r) \times r$ matrix. Inserting the segregated stoichiometry matrix 2.17 into the original dynamics eqn.2.16, we can now see how the set of species $\mathbf{S}(t)$ can be separated into the set of independent species $\mathbf{S}_i(t)$, and the set of dependent species $\mathbf{S}_d(t)$.

$$\begin{aligned}\frac{d}{dt}\mathbf{S}(t) = \frac{d}{dt}\begin{bmatrix} \mathbf{S}_i(t) \\ \mathbf{S}_d(t) \end{bmatrix} &= \begin{bmatrix} \mathbf{N}_r \\ \mathbf{N}_0 \end{bmatrix} \mathbf{v}(\mathbf{S}(t), \mathbf{p}, t), \\ &= \begin{bmatrix} \mathbf{I}_r \\ \mathbf{L}_0 \end{bmatrix} \mathbf{N}_r \mathbf{v}(\mathbf{S}(t), \mathbf{p}, t) \\ &= \mathbf{L}\mathbf{N}_r \mathbf{v}(\mathbf{S}(t), \mathbf{p}, t).\end{aligned}$$

With the separation of the species and stoichiometries into independent and dependent parts, the original dynamics equation can now be partitioned into a pair of equations, describing the dynamics of the independent and dependent parts respectively,

$$\frac{d}{dt}\mathbf{S}_i(t) = \mathbf{N}_r \mathbf{v}(\mathbf{S}(t), \mathbf{p}, t)\ ;\ \frac{d}{dt}\mathbf{S}_d(t) = \mathbf{L}_0 \mathbf{N}_r \mathbf{v}(\mathbf{S}(t), \mathbf{p}, t). \tag{2.18}$$

The independent and dependent parts of eqn.2.18 can be re-combined and upon simplification, we can relate the dynamics of the independent and dependent parts to each other as,

$$\frac{d}{dt}\mathbf{S}_i(t) - \mathbf{L}_0 \frac{d}{dt}\mathbf{S}_d(t) = 0. \tag{2.19}$$

Integrating eqn.2.19, and introducing the vector $\mathbf{T}$ as the constant of integration, we arrive at

$$\mathbf{S}_d(t) - \mathbf{L}_0 \mathbf{S}_i(t) = \mathbf{T}. \tag{2.20}$$

The vector $\mathbf{T}$ is the vector of conserved moieties, and it is uniquely determined by the initial conditions of the chemical species:

$$\mathbf{T} = \mathbf{S}_d(0) - \mathbf{L}_0\mathbf{S}_i(0). \tag{2.21}$$

The combination of eqn.2.20 and eqn.2.21 allow us to uniquely determine the complete set of dependent species values, $\mathbf{S}_d(t)$ in terms of the current independent species values, $\mathbf{S}_i(t)$ and the initial conditions of both the independent and dependent species values, $[\mathbf{S}_i(0), \mathbf{S}_d(0)]$. Therefore, we only need determine $\frac{d}{dt}\mathbf{S}_i(t)$ to fully specify the dynamics of the complete system.

The $\mathbf{L}_0$ matrix can be calculated through a number of standard techniques such as LU factorization, Gaussian elimination or QR factorization. All of these methods are established in [78] and implemented in the LibStruct library [78].

SBML to SBML conversion

The process of applying moiety conservation and model reduction to an existing SBML document can be treated as a mapping of one SBML document to a new SBML document. Syntactically the current SBML specification, [46] allows a conserved system to be fully specified with zero changes required to the SBML syntax specification.

The mapping process is defined by the following procedures, starting with the original document already loaded into the libSBML document object model (DOM):

1. Load the document into libstruct, this will be used to calculate the $\mathbf{L}_0$ matrix and the lists of independent and dependent floating species.

2. Create a new SBML DOM, copy everything with the exception of the floating species from the original into the new DOM.

3. Using the list of $\mathbf{S}_i$ and $\mathbf{S}_d$ copy the floating species from the existing into the new using the ordering specified by libstruct.

4. Create a new set of global parameters corresponding the set of conserved moieties, assign each one of them an SBML id using a UUID, and add them to the new DOM.

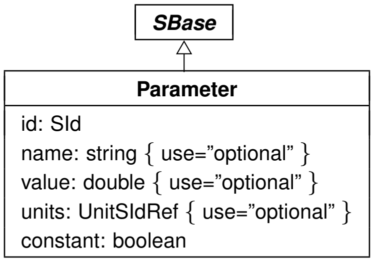


Figure 2.4:

5. Using eqn.2.21, generate a set of assignment rules defining the new conserved moiety variables, $\mathbf{T}$.
6. Using eqn.2.20, generate a set of assignment rules defining the dependent species. The mere presence of an assignment rule will cause the LLVM SBML compiler to interpret a floating species as a dependent floating species which results in no ODE being generated for that species.

The only requirement for specifying a conserved model in SBML is the relaxation of the semantic restriction in section 4.9.3 stating that "there must not be both an AssignmentRule variable attribute and a SpeciesReference species attribute having the same value, unless that species has its boundaryCondition attribute set to "true"". Instead of being considered an error condition, the new semantic interpretation of a species having both a species reference and an assignment rule is to allow the rule to have higher precedence that the reference, i.e. the species is interpreted as a dependent species if it has an assignment rule. Currently in RoadRunner, when the stoichiometry matrix is generated and species is found to have both an a species reference (participates in a reaction) and an assignment rule, an error condition is raised. This can be easily changed so that when a species is found to have both a reference and an assignment rule, no entry in the stoichiometric matrix would be created for product species references if a assignment rule is found. This is a one line change.

Though not absolutely required, we can introduce a new SBML element specialization of the `Parameter` element called `ConservedMoietyParameter`. This new class would not add any new fields to `Parameter`, it is merely there to to inform the SBML compiler not to list these global parameters list of identifiers. We would also add the semantic restriction that `ConservedMoietyParameter` must be defined by an initial assignment rule. Updating the SBML compiler to identify the `ConservedMoietyParameter` and generate the appropriate identifier names for the global parameters and the new conserved moiety parameters, and updating the `SelectionRecord` to identify these new symbols is perhaps a 20 lines of code change.

Mutable Conserved Moieties

During the course of simulation and analysis, frequently, one might need to investigate network properties in relation to the change of conserved moiety values. In general, one might need to change the value of a dependent conserved species. Despite the fact that internally, a subset of the floating species were re-classified as dependent species by the conservation conversion algorithm, the user of the library should ideally not be conserved with such internal details. A user should be allowed to change the value of a species regardless of whether it an independent or dependent species. However, as the dependent species are now no longer part of the state vector, rather they are now defined by rules, it is not possible to change their value as it is possible to change the value of an independent species by changing the value in the state vector. In fact, as mentioned in § 2.4.3, SBML elements that are defined by rules are never allocated any memory storage space – whenever their value is requested, either by an internal SBML dereferencing such as in a rate rule or a function, or externally requested, the actual code that defines the rule is executed to calculate the value. Therefore, in general, it is not possible to modify the value of an element which is defined by a rule. However, in the special case of conserved floating species, we may refer back the exact rule which defines them, eqn. 2.20. With eqn. 2.20, we know that the dependent floating species is explicitly defined as

$$\mathbf{S}_d(t) = \mathbf{L}_0 \mathbf{S}_i(t) + \mathbf{T}. \tag{2.22}$$

It would be possible to change the value of a dependent species by altering the value of an independent species as in general, the $\mathbf{L}_0$ matrix is not square, and hence not invertible.

However, We may also notice that $\mathbf{T}$ is a column vector, and this has the exact same ordering as the dependent species vector $\mathbf{S}_d$. Therefore, if a new value is to be stored in a dependent species, we may calculate the difference between the present value and the new value, and add this difference to the appropriate entry in the $\mathbf{T}$ vector. This may be explicitly stated if the value of x were to be stored in the i^{th} conserved species, $\mathbf{S}_{d,i} \to x$ as

$$\begin{aligned} d &= x - \mathbf{S}_{d,i} \\ T_i &\to T_i + d. \end{aligned} \tag{2.23}$$

The operation specified in 2.23 is currently implemented in the C++ species accessor functions. A limitation of the approach is that currently, SBML events may not alter the value of a conserved species. This limitation will be adressed in a future version as the mutable conserved moiety operation, 2.23 will be moved into the JIT compiled code where SBML event assignments may change these conserved species values.

SBML Extension

libSBML has intrinsic support for extensions, [11, 12]. The majority of new SBML features are first implemented as a plugin to libSBML. The extension system allows one to develop entirely new SBML elements, and using the plugin system of libSBML, documents containing extensions can be read and written directly and transparently.

The moiety conversion algorithm specified here is implemented as an libSBML extension. This means that it may be used independently of the LibRoadRunner library and it can apply the transformation to a document to generate a new moiety conserved document which may be saved to disk or shared between applications.

2.5.4 Spatial Systems using RoadRunner

In this section, I will demonstrate the use of roadrunner used in a spatial environment with simple example of LibRoadRunner used simulate reaction-diffusion models. One of the primary design goals of this library was interoperability with existing simulation libraries, specifically spatial simulators. In order to maximize efficiency, spatial solvers require access to the entire rate of change of the state vector as one single contiguous memory block.

Models described using SBML assume that compartments are well-stirred (CWSC). Chemical species are assumed to distributed instantly and uniformly throughout a finite volume, i.e. the diffusion constant is infinite, or at least significantly larger than the concentration rate of change due to chemical reactions. Though this assumption may hold in certain circumstances, in general, it is relatively rare to find continuous well stirred compartments in nature. Biology is incredibly spatially inhomogeneous [55].

Whilst it is relatively simple to abstract the concept of an ordinary differential equation (ODE) integrator, it is much more difficult to generalize to concept of a partial differential equation (PDE) integrator. Essentially, all one needs to numerically integrate an ODE is a call-back function which returns the rate of change of the state vector. Obviously higher order schemes do use information such as the Jacobian or Hessian, but most numeric integrators only need the rate of change of the state vector. Spatial systems are however defined in terms of PDEs and PDE integrators always require the spatial extents of the system to be described using a mesh or lattice data structure. These data structures vary wildly between different PDE libraries.

Many cellular and tissue simulators such as CompuCell3D [75] or V-Cell [70] incorporate their own PDE integrators. One of the key design goals of LibRoadRunner is interoperability with existing libraries and applications. Therefore, here we will give an example of integrating libRoadRunner into a spatial system, with libRoadRunner providing the rate of change term in a reaction-diffusion system.

Reaction-Diffusion

Most examples in cellular and tissue biology can be described by a reaction-diffusion (RD) equation. Such a system is composed of spatial diffusion term and temporal reaction term,

$$\frac{\partial U}{\partial t} = D\Delta U + R(U), \tag{2.24}$$

where U is a vector of chemical concentrations, D is the (diagonal) diffusion matrix, and $R(U)$ is the rate of change of the chemical concentrations due to local reactions. The

Laplacian operator acting on the chemical field determines the spatial diffusion. RD is simply the Poisson equation with the addition of a reaction term.

Brusselator

The Brusselator (Brussels - Oscillator) is a theoretical model of an auto-catalytic reaction conceived by Prigogine, et al. [57]. Although the Brusselator is a purely theoretical system, it is one of the simplest examples of a cross activator-inhibitor reaction which also includes chlorite-iodide-melonic acid (CIMA), ferrocyanide-iodate-sulphite, and numerous other enzyme catalytic reactions. This well known system is comprised of the following reactions:

$$\begin{aligned}
&A \rightarrow X, && k_1 A \\
&2X + Y \rightarrow 3X, && k_2 X^2 Y \\
&X + B \rightarrow Y + D && k_3 X B \\
&X \rightarrow E && k_4 X, \qquad (2.25)
\end{aligned}$$

where the reaction rates are given by the usual mass-action kinetics. For our purposes, all constants are set to unity. There are two floating species, X, Y, and four fixed boundary species, A, B, D, and E.

In this system, only the two floating species, X, Y have dynamics governed or rate rules, thus these two species comprise the state vector of the system. All other elements of the SBML model are fixed boundary species, parameters or volumes whose values do not change in time.

Discretization

Numerous discretization of the reaction diffusion are possible, however here we will focus on the simplest possible case, the finite forward difference scheme. The forward difference scheme tends to be numerically unstable and is not particularly efficient, however it is extremely simple and thus serves as simple example for numerically solving the RD equations. The forward difference discretization will vary based on what assumptions are made of the

RD system. One might assume that the magnitude of the spatial diffusion term is similar to the reaction rate dynamics, $|D\Delta U| \approx |R(U)|$. Under such assumptions, one might replace the time differential and Laplacian with forward and central difference approximations respectively, and solve the next time step as

$$\begin{aligned}\frac{U_{x,t+\delta t} + U_{x,t}}{\delta t} &= D\frac{U_{x-\delta x,t} - 2U_{x,t} + U_{x+\delta x,t}}{\delta x} + R(U_{x,t}) \\ U_{x,t+\delta t} &= U_{x,t} + D\frac{\delta t}{\delta x}\left(U_{x-\delta x,t} - 2U_{x,t} + U_{x+\delta x,t}\right) + \delta t R(U_{x,t}). \end{aligned} \tag{2.26}$$

One might also assume that the spatial diffusion term is much slower than the reaction term, $|D\Delta U| \ll |R(U)|$. In such cases, the local reaction term can be considered to equilibrate with surroundings faster that the surroundings can change. In such cases, if one used the previous discretization, considerable time can be spent needlessly calculating the Laplacian while using a time step small enough to ensure stability while integrating the reaction term. Therefore, in order to use a suitably large time step to calculate the Laplacian at an appropriate rate, we may choose to integrate the reaction term as an ODE over this time interval as

$$U_{x,t+\delta t} = U_{x,t} + D\frac{\delta t}{\delta x}\left(U_{x-\delta x,t} - 2U_{x,t} + U_{x+\delta x,t}\right) + \int_{t}^{t+\delta t} R(U_{x,t})dt. \tag{2.27}$$

The slow way

perform a reaction-diffusion step

takes the input lattice, ui, performs the reaction - diffusion step, and writes the output to uo.

ui and uo are (nx,ny,2) 3 dimensional arrays, the last index are the 2 elements of the Brusselator state vector.

This is an example of how to write extremely inefficient Python code. The general rule of writing fast code in Python is to write as little Python as possible.

```
def rd_step(ui, uo):
  global time, nx, ny, r, time series, time step
  for i in range(0,nx):
    for j in range(0,ny):
```

```
        # Laplacian in x direction
        uxx = ( ui[i+1 if i < nx-1 else 0, j] - \
          2*ui[i,j] + ui[i-1, j] )/dx2
        # Laplacian in y direction
        uyy = ( ui[i,j+1 if j < ny-1 else 0] - \
          2*ui[i,j] + ui[i, j-1] )/dy2
        # add rate of change due to reaction part
        uo[i,j] = ui[i,j] + dt*D*(uxx+uyy) + \
          dt * r.model.getStateVectorRate(time, ui[i,j])
```

The faster way

A vectorized version of the reaction diffusion step, code is much smaller and runs about 200 times faster than the non-vectorized version above.

This function uses a numpy expression to evaluate the derivatives in the Laplacian, and calculates u[i,j] based on ui[i,j].

- RR getStateVector accepts an N dimensional array, provided the trailing index is the same size as the state vector.
- automatically iterates over leading dimensions.
- best way to write fast code in python is to write as little as possible.
- The rate of change of the state vector is a function of the state vector, hence getStateVectorRate is a const function - does not alter the state of the loaded model.
- Periodic boundary conditions - just a shift of the matrix indices - accomplished with numpy roll.
- ideal way for prototyping new integrators.

```
def rd_step_vec(ui, uo):

  global time, time series, time step, dt, dx2, dy2, dudt

  two_ui = 2.0 * ui
  uxx = (n.roll(ui,1,0) + n.roll(ui,-1,0) - two_ui) / dx2
  uyy = (n.roll(ui,1,1) + n.roll(ui,-1,1) - two_ui) / dy2
  uo[:,:,:] = ui + dt * (D * (uxx+uyy) + \
    r.model.getStateVectorRate(time, ui, dudt))
```

2.6 Community Integration

LibRoadRunner uses libSBML (version 5.9 as of this writing) to read / write SBML documents. This library is also used by the majority of SBML capable applications. Therefore, we can read any document that libSBML is capable of reading. Two features of the SBML L3V1 specification are not currently supported: delay equations and algebraic rules. If either of these features are encountered, they are ignored and warning message is logged and presented.

All our source code is liberally licensed under the un-encumbered Apache Version 2.0 license, so our library may be freely used in the widest range of open source and commercial applications. Our entire development process is transparent, all source code and history is available on the Git Hub source repository at https://github.com/AndySomogyi/roadrunner. We welcome and encourage user contributions. We have created and maintain a LibRoadRunner community web site at http://libroadrunner.org/ where users can participate in the community mailing list to receive help and suggest enhancements. The latest versions of the library and documentation may be downloaded.

As LibRoadRunner is written specifically as a library designed to be used in existing application, we believe this encourages collaboration and involvement with users developing their own uses and applications of our library.

In the relatively short short time since its inception, LLVM has spawned hundreds if not thousands of individual projects. These range from a variety of new languages such as Julia, to new development tools to new applications such as WebC, NaCL. This is in contrast to GCC, which is without doubt an extremely capable compiler, but has not anywhere near number or level of community initiated projects like LLVM.

The reasons for this fact may be in dispute, but likely it is a combination of the fact that LLVM is written in modern C++ rather than C thus making it more accessible; LLVM was whiten from the onset and a modular library rather than an application, or possibly it is the more liberal license, BSD vs. GPL. In any case, we believe that LibRoadRunner has the key attributes which we believe LLVM attractive for community involvement.

Chapter 3

Electrochemical Mitochondrial Modeling

3.1 Introduction

The electrophysiological and metabolic state of heart muscle mitochondria is modeled via a coupled electrometabolomic model. Kinetic metabolic and membrane transport rate equations are coupled via a reaction-electrophysiological model that also accounts for the Debye layer surrounding all membranes via a capacitive equation. The membrane flux potential contributions that affect the membrane potential and another that accounts for the maintenance of charge neutrality far from the membrane are the charge layer that lies within the Debye layer. The model is used to gain insights into the states of heart muscle in healthy and diabetic states.

In a system like a mitochondrion, and in fact most cells, electrophysiology and the metabolomics are strongly coupled that modeling them requires a mathematical framework that accounts for a broad spectrum of processes. For example, most biochemical species are charged and therefore their exchange with the surroundings is strongly affected by membrane potentials. In turn, these biochemical species often mediate activity of the ion pumps that lead to the membrane potentials. Clearly to understand these systems, it is necessary to develop a model that accounts for the charges of biochemical species and its effect on the membrane potential. The objective of the study is to use such a comprehensive electrometabolomic approach to understand the electrometabolomic states of the mitochondrion in normal and diseased systems with the ultimate objective to derive implications for function as it is affected by diabetics.

This chapter will first begin with an overview of the mitochondrial biochemistry, then will proceed to develop a theory to model the chemistry of these electro-chemical kinetics. Once the electro-chemical model has been developed, it will be embedded in a quasi-spatial environment.

The modeling approach taken here will be a Reaction Kinetics (RK) approach. This is the study how the rate of change of a phisical process is relates to other rates of change and the state of the system. This approach differs from the Flux Balance Analysis (FBA) approach which is simpler but less informative. FBA models are significantly simpler that dynamic models develped in RK and require fewer parameters such as free energies, reaction rates or concentrations. However, FBA models provide much less information about the model. FBA assumes the system is operating at a steady state, and is unable to to model any dynamic processes such as oscillations. FBA does not consider the concentrations of species, and an FBA model may contain unrealistic concentrations. FBA may however be usefull in when this model is embedded in a larger spatial system if the characteristic time scale of the subcellular network is much faster than the characteristic time scale of the spatial process.

Chemical and biochemical redox reactions can all, in principle, be carried out by transferring the electrons from the molecule being oxidized to an electrode located in one solution, and then delivering electrons to the molecule being reduced via another electrode located in a separate solution. I

Oxidation reduction reactions transfer electron (or other charged group, i.e. hydride H^-, Hydroxide, OH^{-1}, etc) from electron donor (reducing agent) to a electron acceptor (oxidizing agent). The reducing agent loses an electron(s) and changed from a reduced to an oxidized form. The oxidizing agent accepts an electron and changes from the oxidized to the reduced form. The reducer and oxidizer pair form a redox couple.

The standard reduction potential, E^0 is also referred to as the midpoint potential of a redox couple. E^0 is defined as the voltage at which the concentrations of oxidized and reduced products are at equilibrium. E^0 is a measure of the strength of an electron donor, i.e. a component with a more negative E^0 is a stronger electron donor or equivalently, a weaker electron acceptor. When two half equations are combined, the E^0 potential determines the direction the electrons will flow, in the absence of any external potentials or concentration gradients.

3.2 Mitochondrion Structure and Function

Mitochondria, found in most eukaryotic cells, are organelles that produce most of the energy supply for the cell in the form of ATP. The mitochondrion's double-membrane organization allows for five separate compartments within the organelle including the outer and inner mitochondrial membranes, the inter-membrane space (the space between the inner and outer membranes), the cristae (formed by the inner membrane foldings), and the matrix (the space within the inner membrane). The mitochondrion serves several functions, such as fatty acid and glycogen metabolism. Most importantly, mitochondria provide cellular respiration for cells by coupling oxidative phosphorylation and membrane potential. Glycolysis, which occurs in the cytosol, produces pyruvate that is converted into carbon dioxide and water via the citric acid cycle in the matrix. NADH produced from the citric acid cycle is oxidized in the electron transport chain (ETC), located in the cristae, to establish a proton gradient, or membrane potential, across the inner membrane. The proton gradient drives protons to go through the ATP synthase of the inner membrane, powering the synthase to phosphorylate ADP to ATP.

Mitochondria perform the final stages of the cellular respiration process. Cellular respiration at a very high level is the transduction of energy stored in the chemical bonds of glucose $C_6H_{12}O_6$ to phosphorylate ADP into ATP. ATP is the principal source of energy for all cellular processes. In respiration, glucose is oxidized and oxygen is reduced to form water. The carbon atoms of the sugar molecule are released as carbon dioxide CO_2. Cellular respiration proceeds in two major steps: 1) glycolysis and 2) aerobic respiration. Anaerobic respiration also occurs, but will not be discussed here. The glycolysis pathway produces two ATP, and if O_2 is available, thirty-four more ATP are produced in the aerobic pathway.

Oxidative phosphorylation is the final stage of aerobic oxidation in eukaryotes and occurs in the mitochondrion. Here, the reduced coenzymes NADH and $FADH_2$ which were produced by 1) aerobic oxidation of pyruvate by the citric acid cycle, and 2) oxidation of fatty acids and amino acids, are oxidized in the ETC in order to establish a proton gradient across the mitochondrial matrix-intermembrane space. The potential energy in the proton gradient is then used by ATP synthase to produce ATP. Here, protons fall back down the proton gradient through the ATP synthase ion channel. The energy from the proton exchange is used to phosphorylate an ADP back into an ATP.

3.2.1 Electron Transport Chain

The ETC is a series of sequential oxidation-reduction reactions which pass electrons from NADH or $FADH_2$ through a sequence of protein complexes and charge carriers, finally O_2, producing H_2O. The ETC is composed of four protein complexes, I, II, III, IV, in the matrix-intermembrane space membrane. These complexes sequentially catalyze redox reactions. Electrons are transferred to O_2 which is finally reduced to H_2O. The flow of electrons through the chain is thermodynamically favorable because each subsequent carrier has a higher electron affinity than the previous one. Hence, the electron flow is spontaneous. The energy obtained from each redox reaction is used to pump protons from the mitochondrial matrix into the inner membrane space. Complex I, Complex III and Complex IV pump protons across the inner mitochondrial membrane.

Electrons first enter the ETC from NADH at Complex I. FAD_2 electrons enter at complex II. These electrons are then passed from Complex I or II to charge carrier Coenzyme Q10 , commonly referred to as CoQ10 (Co indicates it is a coenzyme, Q indicates quinone chemical group, and 10 is the number of isoprenyl subunits in its tail), here we will simply use Q as this is the only quinone we will refer to. Q is a hydrophobic (fat-soluble) molecule, and is therefore mobile in lipid plasma membranes. Q freely diffuses inside the membranes, but is typically not found in any significant quantities in any compartment. Q ferries electrons from complex II to complex III. Electrons at complex III are transferred to cytochrome C. Cytochrome C is a highly hydrophilic (water soluble) protein, unlike other cytochrome, hence it is found freely diffusing in the compartments rather than the lipid membranes. Electrons are then ferried by cytochrome to complex IV. Finally, electrons are transferred by complex IV to O_2 is then reduced to water.

Each protein complex uses metal containing prosthetic groups such as iron-sulfur clusters, copper ions, hemes or flavins as electron carriers. The flavins are FMN - $FMNH_2$ in complex I or FAD - $FADH_2$ in complex II.

Complex I, also called NADH-ubiquinone oxidoreductase is a large L-shaped protein. It is the largest of the ETC complexes, with a mass of $\approx 850\,\mathrm{kDa}$ [53]. Complex I contains flavin mononucleotide (FMN) as the NADH oxidation active site, at least one Q binding site, and possibly eight iron-sulfur (FeS) clusters. The primary role for complex I is to catalyze a pair of coupled processes. The first is the exergonic transfer of hydride ion from NADH and a proton from the matrix to FMN, expressed as

$$\mathrm{NADH} + \mathrm{H}^+ + \mathrm{Q} \longrightarrow \mathrm{NAD}^+ + \mathrm{QH}_2. \tag{3.1}$$

The free energy obtained from this reaction is used by the second process which is the endergonic transfer of four matrix protons to the the inner membrane space. The overall reaction here is

$$\mathrm{NADH} + 5\,\mathrm{H}_\mathrm{m}^+ + \mathrm{Q} \longrightarrow \mathrm{NAD}^+ + \mathrm{QH}_2 + 4\,\mathrm{H}_\mathrm{i}^+, \tag{3.2}$$

where the m subscript is used to indicate matrix molecules and the i subscript indicates inner membrane molecules.

The FMN site and the FeS clusters exist in the Iλ subunit of Complex I. NADH can bind to the FMN site on the Iλ subunit and transfer a hydride (2 electrons) to the FMN. The electrons travel through to the FeS clusters and the H^+ is separated and falls off into the matrix along with the NAD^+. When looking as the whole Complex I, this H would be re-absorbed along a site on the membrane arm of Complex I along with another H^+ and would join with the pair of electrons to be combined with a Q to form QH_2. Therefore, if we look at just the net reaction occurring in the Iλ subunit, we see that it can be expressed as

$$\mathrm{NADH}_\mathrm{m} \longrightarrow \mathrm{NAD}_\mathrm{m}^+ + \mathrm{H}_\mathrm{m}^+ + 2\,\mathrm{e}_\lambda^-, \tag{3.3}$$

where the λ subscript is used to indicate the that the electrons are deposited at the membrane facing side of the Iλ subunit. The correct simulation of this reaction will be the key purpose of this chapter.

It has been show that in general, the complex I mediated interconversion of NADH and

NAD^+ 3.3 is in general a reversible reaction [7,16,17,82]. The free energy of reaction ΔG of each component of reaction 3.3 is known, there is a level of uncertainty of the exact values of the forward and reverse rate constants and the exact form of the reaction rate rule itself.

This chapter will focus on using the tools developed in chapter 2 to develop a physically based simulation of Iλ subunit of mitochondrial ETC complex I in order to develop estimates of the forward and reverse rates of reaction 3.3.

3.3 Quantitative Model Formalism

The development of mitochondrial ETC model begins with a physical description of the system. A schematic of the system described here is depicted in 3.1. Although this section will focus on the NAD Oxidoreductase protein, the following discussion is completely general and is valid for any trans-membrane protein complex. This section will first develop the formalism to describe a general electrochemical system in the absence of of any mobile electrolytes, then will add the physical realism of accounting for mobile electrolytes which are present in any biologically relevant regime.

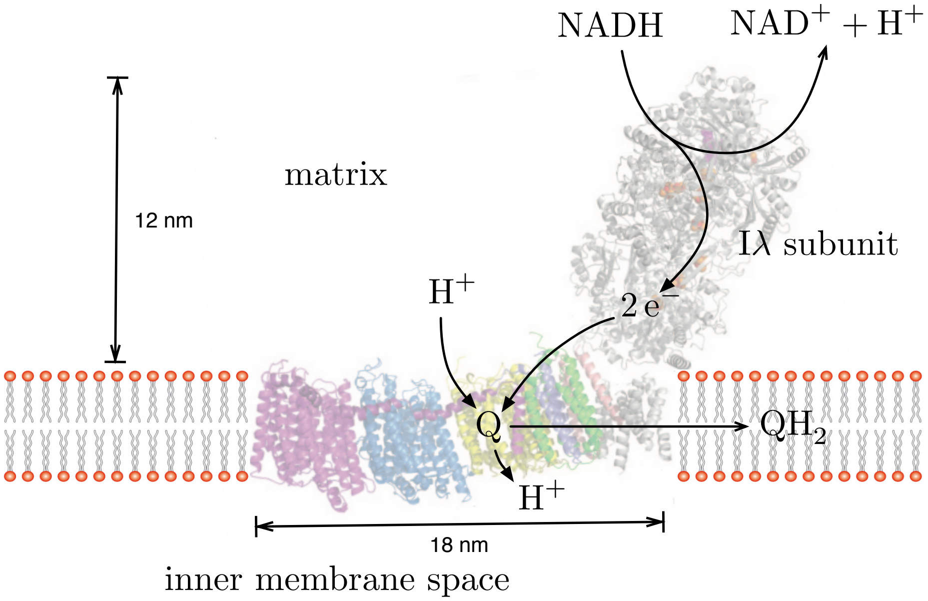


Figure 3.1: Mitochondrial Complex I, NAD Oxidoreductase. This large "L" shaped membrane protein exists in the mitochondrial matrix - inner membrane space plasma membrane. The upper arm extends ≈ 12 nm into the matrix. Adapted from [26].

3.3.1 Reaction Rate Kinetics

Reaction rate kinetics describe the concentration of reacting entities over time. They are most commonly used to describe the rate of change of chemical compounds in solution, however this approach can be used to describe almost any system with interacting time varying concentrations of some quantity: numbers of bacteria, numbers of active and quiescent neurons in a cluster, number of trees and parasitic insects in a forest, and so forth.

Biological processes nearly always occur at constant temperature and pressure. Thus they can be characterized by the change in Gibbs free energy of reaction, ΔG per process. $-\Delta G$ may also be known as the chemical affinity [49, 54] in certain contexts, however most modern texts generally use the term free energy of reaction. This function indicates the maximum amount of useful work which may be extracted from a system held at constant temperature and pressure. The sign and magnitude of the change in Gibbs Free Energy determines whether a process will occur spontaneously if at all.

ΔG is a measure of how far the system is from equilibrium. At equilibrium, ΔG is zero, and the farther a system is from equilibrium, the larger the magnitude of ΔG is, and thus, the greater is the amount of work that the process can perform, or the amount of works performed on the process. If $\Delta G < 0$, the reaction proceeds spontaneously. If $\Delta G = 0$, the reaction is at equilibrium, and if $\Delta G < 0$, the reaction proceeds spontaneously in the reverse direction. Formally, ΔG is written as a stoichiometric weighted sum of the chemical potentials of the reaction reactants. All reactions are in principle reversible, so accounting for the products and reactants on both sides, ΔG is written as a stoichiometric weighted sum of the individual chemical potentials,

$$\Delta G = \sum^{p} \nu_p \mu_p(T) - \sum^{r} \nu_r \mu_r(T) - \tag{3.4}$$

where the sums, r, p are over the reactants and products respectively, and ν are the stoichiometric coefficients. Here, the chemical potential, μ is written as

$$\begin{aligned}\mu_i(T) &= \mu_i^* + RT\ln(a_i) \\ &\approx \mu_i^* + RT\ln(c_i),\end{aligned} \tag{3.5}$$

where μ_i^* is the reference chemical potential, and the activity a has been approximated with the chemical concentration c.

For a reaction to be completely irreversible, the free energies of formation of the products would have to be negative infinity. This may occur when either the enthalpy of formation tends to negative infinity or the entropy of formation tends to infinity. The former is perhaps unlikely in nature, however entropy tending to infinity may occur in situations where the resulting products become separated by distance or barrier such as say a trans-membrane ion pump.

When the number of interacting particles is sufficiently large and the reaction rate constants is sufficiently low, reaction rate systems may be solved approximately using the law of mass action, which states that the rate of any given chemical reaction is proportional to the product of the *activities* of the reactants. Activities however are not usually known for most biological conditions, so they are typically approximated with the chemical concentrations. The law of mass action states that if a item A reacts with item B to produce item C with the reaction

$$A + B \overset{k}{\rightarrow} C, \tag{3.6}$$

then the reaction rate is kAB, with k as the forward rate constant, and item concentrations denoted by A and B. The rate of production or consumption of any item must equal the reaction rate:

$$\frac{dC}{d\tau} = -\frac{dA}{d\tau} = -\frac{dB}{d\tau} = kAB. \tag{3.7}$$

The reaction rates are a measure of how fast a reaction can occur, it is the product of the reaction rate constant, k in this case, and an expression which approximates the availability of the reactants, AB in this case. The reaction rates as determined by the law of mass action are approximations, and valid only near equilibrium conditions. In general, with a system of multiple reactants, and both a forward and backward reaction rate:

$$\alpha A + \beta B + \cdots \underset{k_-}{\overset{k_+}{\rightleftharpoons}} \sigma S + \tau T + \cdots \ , \tag{3.8}$$

the forward and backward reaction rates ν_f, ν_b are approximated as

$$\begin{aligned} \nu_f &= k_+ A^\alpha B^\beta C^\gamma \cdots \\ \nu_b &= k_- S^\sigma T^\tau U^\upsilon \cdots . \end{aligned} \tag{3.9}$$

The law of mass balance states that the rate of change of a substance must be equal to rate of production minus the rate of consumption of the substance, or more formally:

$$\frac{dS}{dt} = \sum production - \sum consumption. \tag{3.10}$$

Thus from (3.6), we can see that both A and B are being consumed at the rate kAB, and C is being produced at that rate.

The forward and reverse reaction rates, according to the law of mass action are given as

$$R^+ = k^+ \prod_r a_r^{\nu_r} \ , \ R^- = k^- \prod_p a_p^{\nu_p}, \tag{3.11}$$

where a is the molar activity, and p and r are the products and reactants respectively of a given reversible reaction. The free energy of reaction, ΔG° is the stoichiometric weighted sum of the free energies of formation,

$$\Delta G^\circ = \sum_p \nu_p \Delta G_f^\circ[p] - \sum_r \nu_r \Delta G_f^\circ[r]. \tag{3.12}$$

The law of mass action can be made formal by looking at the definition of the free energy of reaction, 3.4. At equilibrium, $\Delta G = 0$, inserting 3.5 into 3.4, expanding the sums and grouping the reference chemical potentials, we have

$$\begin{aligned}\Delta G|_{eq} &= \sum^{p} \nu_p \mu_p(T) - \sum^{r} \nu_r \mu_r(T) = 0 \\ &= \sum^{p} \nu_p \mu_p^*(T) + RT \sum^{p} \nu_p \ln(c_{p,eq}) - \sum^{r} \nu_r \mu_r^*(T) - RT \sum^{r} \nu_r \ln(c_{r,eq}) = 0 \\ RT \sum^{p} \nu_p \ln(c_{p,eq}) - RT \sum^{r} \nu_r \ln(c_{r,eq}) &= \sum^{r} \nu_r \mu_r^*(T) - \sum^{p} \nu_p \mu_p^*(T). \end{aligned} \tag{3.13}$$

Dividing both sides by RT and exponentiating, one arrives at the definition for the equilibrium constant,

$$K = \frac{k^+}{k^-} = \frac{\prod^p c_{p,eq}^{\nu_p}}{\prod^r c_{r,eq}^{\nu_r}} = \exp\left(\frac{\sum^r \nu_r \mu_r^*(T) - \sum^p \nu_p \mu_p^*(T)}{RT}\right) = \exp\left(\frac{-\Delta G^\circ}{RT}\right). \tag{3.14}$$

The equilibrium constant K also gives the ratio of the forward and reverse rate constants. Here, we may define the stoichiometric weighted sum of the standard chemical potentials as the standard change of reaction in Gibbs free energy as ΔG°.

Away from equilibrium, ΔG is not zero, it my be written as 3.13 with the equilibrium concentrations replaced with the current concentrations. The sum of the standard chemical potentials is the same as in 3.13 and may be replaced with $-\Delta G^\circ$ as

$$\begin{aligned}\Delta G &= \sum^{p} \nu_p \mu_p^*(T) + RT \sum^{p} \nu_p \ln(c_{p,eq}) - \sum^{r} \nu_r \mu_r^*(T) - RT \sum^{r} \nu_r \ln(c_{r,eq}) \\ &= \Delta G^\circ + RT \left(\sum^{p} \nu_p \ln(c_p) - \sum^{r} \nu_r \ln(c_r) \right) \\ &= \Delta G^\circ + RT \ln\left(\frac{\prod^p c_p^{\nu_p}}{\prod^r c_r^{\nu_r}}\right) \\ &= -\left(RT \ln(K) + RT \ln\left(\frac{\prod^r c_r^{\nu_r}}{\prod^p c_p^{\nu_p}}\right)\right) \\ &= -RT \left(\ln\left(\frac{k^+}{k^-}\right) + \ln\left(\frac{\prod^r c_r^{\nu_r}}{\prod^p c_p^{\nu_p}}\right)\right) \\ &= -RT \ln\left(\frac{k^+ \prod^r c_r^{\nu_r}}{k^- \prod^p c_p^{\nu_p}}\right). \end{aligned} \tag{3.15}$$

Finally, using the definition of the forward and reverse reaction rates 3.11, we may relate these rates to the Gibbs free energy change of reaction as

$$\Delta G = -RT \ln \left(\frac{R^+}{R^-} \right). \tag{3.16}$$

The previously developed formalism can describe a very wide range of reactions, however it does not account for reactions involving charge transfer (electron transfer). A significant portion of biological reactions are oxidation or reduction reactions and do involve the transfer of electrons. Any redox reaction can in principle be carried out by transferring electrons from a molecule being oxidized in one solution to molecule being reduced in a separate solution through an electrode. Frequently in nature, when when the electrode takes the form of a trans-membrane protein and the oxidized and reduced molecules are in solutions separated by a membrane, this is exactly what occurs. In redox reactions, the potential difference strongly influences the rate of reaction. This influence is described with the addition of an electrical term to the chemical potential, creating the *electrochemical potential*,

$$\tilde{\mu}_i = \mu_i + z_i F \phi, \tag{3.17}$$

where z_i is the electron valence, F is Faraday's constant, and ϕ is the potential difference. In order to determine how the electrical potential difference influences the reaction rates, we will begin with the introduction of an augmented Gibbs free energy of reaction term, $\Delta\tilde{G}$, which is defined as a stoichiometric weighted sum of electro-chemical potentials instead of chemical potentials. They carrying out the procedure outlined above, we will derive a set of augmented reaction rates.

The augmented Gibbs free energy is defined similar to the Gibbs free energy of reaction 3.4 as

$$\begin{aligned} \Delta\tilde{G} &= \sum^{p} \nu_p \tilde{\mu}_p(T) - \sum^{r} \nu_r \tilde{\mu}_r(T) && (3.18) \\ &= \Delta G + \sum^{p} \nu_p z_p F \phi - \sum^{r} \nu_r z_r F \phi. && (3.19) \end{aligned}$$

The summation over the first part of the chemical potential, and the stoichiometric sum was carried out on the electrical part. A pair of augmented reaction rates may be defined similarly to 3.4 as

$$\Delta\tilde{G} = -RT\ln\left(\frac{\tilde{R}^+}{\tilde{R}^-}\right). \tag{3.20}$$

Combining 3.19 and 3.20, we may solve for the augmented reaction rates as

$$\begin{aligned}
-RT\ln\left(\frac{\tilde{R}^+}{\tilde{R}^-}\right) &= \Delta\tilde{G} \\
&= \Delta G + \sum^{p}\nu_p z_p F\phi - \sum^{r}\nu_r z_r F\phi \\
&= -RT\ln\left(\frac{R^+}{R^-}\right) + \sum^{p}\nu_p z_p F\phi - \sum^{r}\nu_r z_r F\phi \\
\frac{\tilde{R}^+}{\tilde{R}^-} &= \frac{R^+}{R^-}\exp\left(\frac{\phi F}{RT}\underbrace{\left(\sum^{r}\nu_r z_r - \sum^{p}\nu_p z_p\right)}_{n}\right) \\
&= \frac{R^+}{R^-}\exp\left(\frac{\phi F}{RT}n\right).
\end{aligned} \tag{3.21}$$

It may be observed that the sum over the reactant and product stoichiometric coefficient in 3.21 is the total number of electrons transferred in the reaction. This value depends only on the constant electronic valences and stoichiometric coefficient, and may be replaced with n.

At this point, we may notice that there is no unique way to determine what fraction of the exponential correction factor should belong to the forward or reverse reaction rates. Therefore, we introduce a *transfer coefficient* α, such that for any number x, $x = (1+\alpha-\alpha)x$, hence, $x = \alpha x + (1-\alpha)x$. Re-writing the exponential term with α,

$$\begin{aligned}
\frac{\tilde{R}^+}{\tilde{R}^-} &= \frac{R^+}{R^-}\exp\left(\alpha\frac{\phi F}{RT}n + (1-\alpha)\frac{\phi F}{RT}n\right) \\
&= \frac{R^+}{R^-}\frac{\exp\left(\alpha\frac{\phi F}{RT}n\right)}{\exp\left(-(1-\alpha)\frac{\phi F}{RT}n\right)}.
\end{aligned} \tag{3.22}$$

Therefore,

$$\tilde{R}^+ = R^+\exp\left(\alpha\frac{\phi F}{RT}n\right)\ ;\ \tilde{R}^- = R^-\exp\left(-(1-\alpha)\frac{\phi F}{RT}n\right). \tag{3.23}$$

For an electrochemical reaction to be in equilibrium, the net reaction rates, $\tilde{R}^+$ and $\tilde{R}^-$

much be equal. In a redox reaction which involves the transfer of electrons, may in general proceed spontaneously without external potential, ϕ. This external potential, frequently referred to as the stopping potential is written as E^0. Regardless of whether the system is at equilibrium or not, the relationship of the change in free energy, ΔG to the forward and reverse reaction rates specified by 3.16 always holds. In order to determine the relationship between E^0 and ΔG°, we may set 3.21 equal to 1, and using 3.16, we arrive at

$$\Delta G = nF\phi \text{ , or } \Delta G^\circ = -nFE^0 \tag{3.24}$$

at equilibrium. Here, n is the number of electrons transferred per reaction, and F is Faraday's constant.

E^0 is readily accessibly from experimental data, however at this point, there is still some ambiguity as to how exactly to specify the forward and reverse rates, k^+ and k^- respectively. ΔG° only specifies the ratio of k^+ to k^-, not their individual values. Thus, we may introduce an additional parameter, called the *standard rate constant*, which specifies the magnitude of each rate constant and removes this ambiguity. Starting with 3.14, we may define k° such that,

$$K = \frac{k^+}{k^-} = \frac{k^\circ}{k^\circ}\frac{k^+}{k^-}. \tag{3.25}$$

k° is a parameter which typically has units of $^{\text{cm}}/_{\text{seconds}}$, and does not affect the equilibrium value of the reaction (which is uniquely specified by ΔG° from equilibrium thermodynamics), but rather specifies *how fast* the reaction will approach equilibrium. k° may also be referred to as the "kinetic facility" of a redox couple. A reaction with a large k° (0.1 to 10 cm/s) will tend to equilibrium faster than reactions with a small k°.

Using the previously developed relations of K and ΔG°, we may fold these two values into 3.23, and use the single previously developed parameter α to uniquely determine these values. Starting with 3.21, we expand the standard reaction rates into their mass action products and rate constants, then using 3.14 and 3.24, we have

$$\begin{aligned}
\frac{\tilde{R}^+}{\tilde{R}^-} &= \frac{R^+}{R^-}\exp\left(\frac{\phi F}{RT}n\right). \\
&= \frac{k^\circ k^+ \prod_r c_r^{\nu_r}}{k^\circ k^- \prod_p a_p^{\nu_p}}\exp\left(\frac{\phi F}{RT}n\right) \\
&= \frac{k^\circ \prod_r c_r^{\nu_r}}{k^\circ \prod_p a_p^{\nu_p}}\exp\left(\frac{-\Delta G^\circ}{RT}\right)\exp\left(\frac{\phi F}{RT}n\right) \\
&= \frac{k^\circ \prod_r c_r^{\nu_r}}{k^\circ \prod_p c_p^{\nu_p}}\exp\left(\frac{nF}{RT}\left(\phi + E^0\right)\right). \qquad (3.26)
\end{aligned}$$

Finally, re-applying the process of introducing the α parameter in 3.31, we arrive at

$$\tilde{R}^+ = k^\circ \prod_r c_r^{\nu_r}\exp\left(\alpha\frac{nF}{RT}\left(\phi + E^0\right)\right)\ ;\ \tilde{R}^- = k^\circ \prod_p c_p^{\nu_p}\exp\left(-(1-\alpha)\frac{nF}{RT}\left(\phi + E^0\right)\right). \qquad (3.27)$$

This rate equation is effectively identical to the well accepted Butler-Volmer equations [18,29]. This formalism completely eliminates the forward and reverse rate constants, k^+ and k^- which can be an very large source of uncertainty, and may be very difficult to estimate from experimental data as they effect the reaction is non-trivial ways. These pair of parameters are replaced them with known physical constants and a pair of parameters which may be easily fit from experimental data [80].

The difference between the forward and reverse rates from 3.27 specifies the net rate of reaction, or *flux* of a chemical reaction in units $^{\text{mol}}/_{\text{literseconds}}$. In other words, it specifies how fast the concentration of a given set of substances (reactants) is being converted into a set of different substances (products). If this reaction is is a redox reaction occurring at the face of an electrode, membrane, or some other surface, then it also specifies the amount of electrical current flowing through that surface. This electrical current is given by

$$\begin{aligned}
J &= nFAk^\circ\left(\tilde{R}^+ - \tilde{R}^-\right) \\
&= nFAk^\circ\left(\prod_r c_r^{\nu_r}\exp\left(\alpha\frac{nF}{RT}\left(\phi + E^0\right)\right) - \prod_p c_p^{\nu_p}\exp\left(-(1-\alpha)\frac{nF}{RT}\left(\phi + E^0\right)\right)\right),
\end{aligned} \qquad (3.28)$$

where n is the number of electrons transferred per reaction, and A is the area of the surface.

Physical Interpretation of Kinetic Law Parameters

The electric flux with varying α parameters is plotted in 3.2. This section will summarize this subsection with discussion of the physical meaning of the parameters that were introduced.

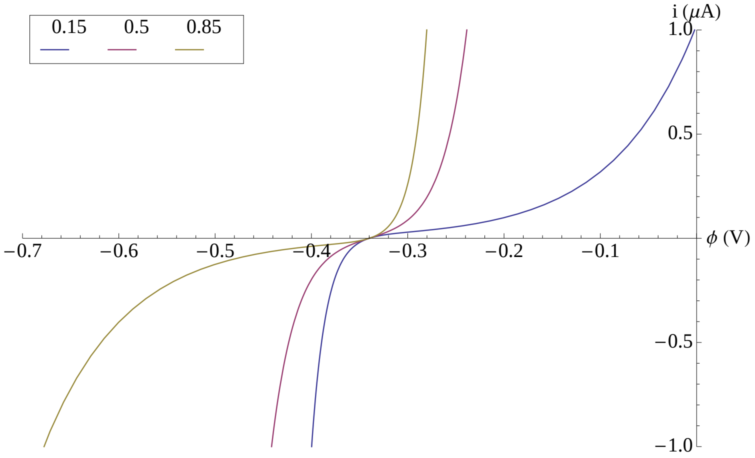


Figure 3.2: The effect of varying the transfer coefficient, α on the electric flux 3.56 as a function of potential, 3.56. Here, $n = 2$, $A = 1\,\text{cm}$, $k^\circ = 10^{-7}\,\text{cm/s}$, and the oxidation and reduction concentrations were held constant at $1\,\text{mM/liter}$, and $E^\circ = -3.2$

In the electric flux kinetic law, 3.56, the simplest parameter is the standard, or *midpoint potential*, E°. This is the potential where half of the molecules are in the oxidized state, and the other half are in the reduced state, it is the midpoint of a redox titration. This is the Nernst potential which is required to hold the reaction at equilibrium, and this is the only parameter which is may be calculated through equilibrium thermodynamics. In this case, it also represents the potential zero crossing of the current flow.

The standard rate constant, k° is a measure of how fast reaction will take place, it is a measure of the *kinetic facility* of the redox couple. Simple redox couples (such as those involving a single electron transfer) will tend to have large k° values, on the order of $0.01 - 10\,\text{cm/s}$. More complex reactions (multiple electron transfers) will tend to have smaller k° values, the slowest k° values are on the order of $10^{-10} - 10^{-9}\,\text{cm/s}$.

The transfer coefficient α, also called *symmetry coefficient* is a is a measure of the the symmetry between the forward and reverse electron transfer reactions – it is a measure of the symmetry of the energy barrier. The variation of α with all other parameters held constant is depicted in fig. 3.2. An α value of 0.5 means that the reaction is perfect symmetric about the midpoint potential. Effectively, α is a measure of how steep the forward or reverse energy barriers are. The closer α is to one, the more irreversible the reaction becomes. On examination of fig. 3.2, we may see that as $\alpha \to 1$, a very small potential may drive a comparatively large forward current, but increasing amounts of potential are required to drive a reverse current. Similarly, as $\alpha \to 0$, small negative potentials result in large reverse current flow, very large forward potential is required to drive a forward current.

Fig.3.2 is an example of a *voltammogram*. This type of plot is readily produced in a *cyclic voltammetry* experiment.

In the following section, we will introduce the *cyclic voltammetry* experiment and show how the results of these experiments can be used to fit the α parameter and verify the E^0 constant.

3.3.2 Interpretation of Experimental Data

Cyclic Voltammetry

Cyclic voltammetry (CV) is a commonly used experimental technique that can provide significant insight in the understanding of electro-chemical redox reactions. A CV experiment is a conceptually very simple experiment which readily be performed on any laboratory bench top with minimal set of laboratory equipment. This, combined with the utility has led to a widespread use of CV for investigation of a wide variety of redox systems [6,20,34].

A typical CV experiment, a *potentiostat* sweeps the potential of a working electrode (WE) and measures the resulting current between the WE and the reference electrode (RE). The potential / current relationships are then plotted in a voltammogram such as in fig. 3.4. The RE This is usually made of an inert metal (such as Gold or Platinum). The WE is is typically made of, or coated with the a component of the redox couple being studied. The solution usually a liquid with a high dielectric constant (e.g. water). A background electrolyte such as an electro-chemically inert salt (e.g. NaCl or Tetra butylammonium perchlorate, TBAP) and is usually added. The solution also must contain the reactant

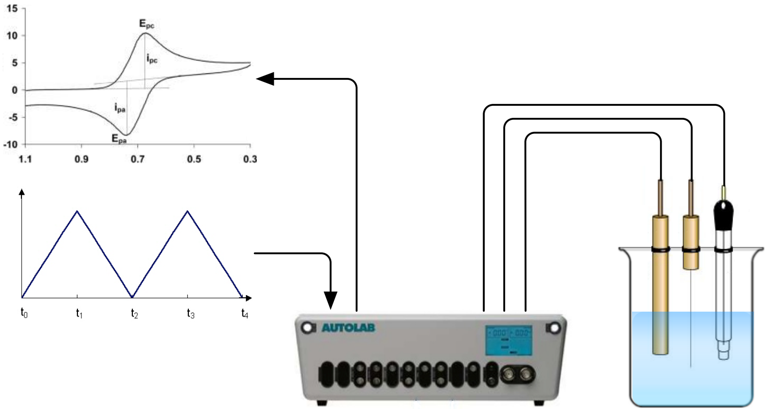


Figure 3.3: A schematic of a cyclic voltammetry experiment as discussed in 3.3.2.

being studied, typically in low concentration 10^{-3} M.

In order to maintain a constant potential whilst passing current to counter the redox events occurring at the working electrode, a third auxiliary electrode which passes all the current needed to balance the current observed at the working electrode.

Traditionally voltammetry experiments are used to investigate the behavior of redox couples where the first component of the couple is in solution and the other component is the WE. Relatively recently (1997), Armstrong, Heering and Hirst pioneered *protein film voltammetry*, (PFV) in which the surface area of the WE is coated with a electro-catalytic protein. Here, the enzyme is adsorbed onto the electrode surface and there is a direct electron transfer. Electrons flow between the enzyme and the electrode, via the active site of the enzyme when the electrode potential is appropriate ($\phi \neq E^{\circ}$).

A typical cyclic voltammogram is show in fig. 3.4. Experimental CV voltammograms may frequently exhibit complex behavior, however all of them tend to show a limiting current flow in the high and low potential limits. One may immediately notice that this experimental curve is markedly different from the theoretical curve shown in fig. 3.2. The theoretical curve shows no limiting behavior at the potential limits. The key reason for this is that the theoretical curve does not take into account any mass (substrate or reactant) limiting behavior.

The limiting behavior seen in fig. 3.4 is due the development of a *depletion layer* on

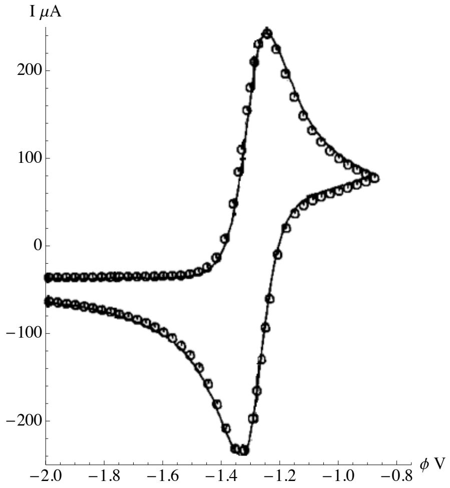


Figure 3.4: Experimental cyclic voltammogram of the ${O_2}^{+}e \longleftrightarrow {O_2}^{-}$ reaction Scan rate: 100 V/s. Taken from [30].

the electrode surface [30]. As the redox potential increases farther away from the Nernst potential, the reaction rate increases exponentially. This causes a rapid consumption of the reactants in thin layer surrounding electrode face where the reaction is occurring. Once all of the reactants have been consumed in this region, any additional reactant can only appear in this region as the result of passive diffusion from the bulk to the depletion layer.

This behavior may be modeled by embedding the redox reaction in a depletion compartment of finite size. The redox couple reactants / products are only consumed produced into this depletion compartment, and the are transferred to / from the electrode. An additional set of diffusion reaction may be introduced to allow passive diffusion between the bulk and depletion compartments. A schematic of the depletion and bulk compartments are presented in fig. 3.5. With the addition of the bulk and depletion compartments, the set of reactions now becomes.

$$\begin{aligned}
&\mathrm{Re_d} + \mathrm{e} \longleftrightarrow \mathrm{Ox_d} \\
&\mathrm{Re_b} \overset{k_d}{\longleftrightarrow} \mathrm{Re_d}\ ,\ k_d = -D_{Re}([\mathrm{Re_d}] - [\mathrm{Re_b}]) \\
&\mathrm{Ox_b} \overset{k_d}{\longleftrightarrow} \mathrm{Ox_d}\ ,\ k_d = -D_{Ox}([\mathrm{Ox_d}] - [\mathrm{Ox_b}])
\end{aligned} \tag{3.29}$$

Here, the passive diffusion is conveniently introduced as an additional pair of trans-compartment

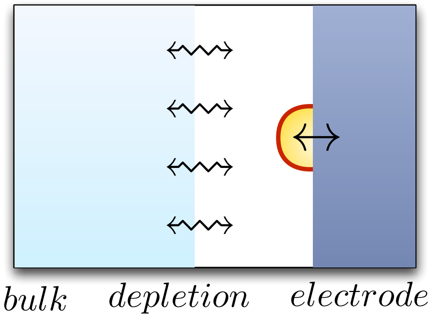
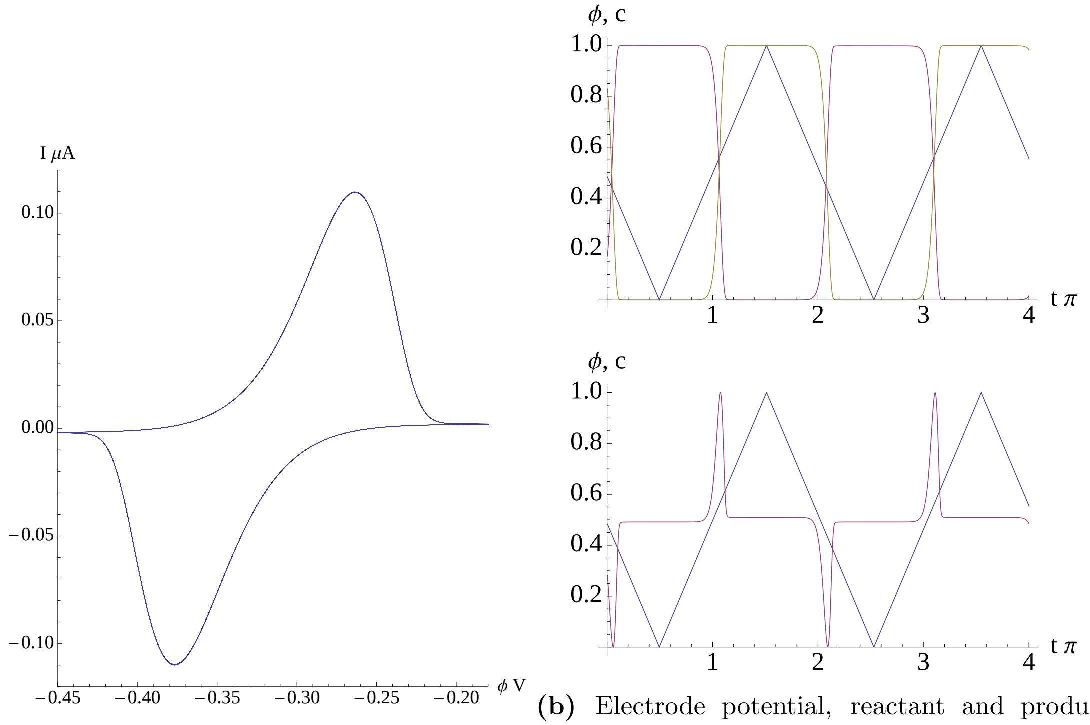


Figure 3.5: The addition of a bulk compartment where all species concentrations are maintained at fixed values, a small depletion layer compartment, and the working electrode.

reactions, with the rate given by Fick's law 3.60, and the rate of the redox reaction given by 3.27.

The voltammogram from a redox reaction which includes the depletion layer compartment, 3.6a generally corresponds well to typical experimental data 3.4. In such a system, the concentration of the reactants is rapidly consumed by the reaction as the electrode potential deviates from the Nernst potential. The current peaks here occur on each of the Nernst potential then rapidly fall off to the maximal rate permitted by passive diffusion from the bulk to the depletion compartments. This may seem counterintuitive, but consider that just as the potential sweeps from just below to just above the Nernst potential, the reaction is switching direction from proceeding in the reverse direction to proceeding in the forward direction. In the reverse direction, the concentration of the reactants was replenished both from the redox reaction itself and from diffusion from the bulk. Hence, the concentration of the reactant is at is highest just as the potential is transitions. Thus, the current peaks correspond to the location of the maximal reactant availability and electrode potential high enough the drive the reaction. The normalized current, potential, and reactant, product concentrations are shown in 3.6b.

The behavior of voltammogram for simple redox reactions is however very dissimilar to the highly complex electro-catalytic reactions such as NADH oxidoreductase which is the topic of this chapter. The structure of simple redox reactions is well known and corresponds well to the formalism developed here. The behavior exhibited in the voltammograms for more complex electro-catalytic enzymes such as NADH oxidoreductase is markedly differ-



(a) CV diagram

(b) Electrode potential, reactant and product concentrations and current.

Figure 3.6: Simulation of a redox reaction with depletion layer. The addition of a depletion layer compartment results in qualitative agreement with experimental data such as 3.4. Potential, ϕ is blue, concentrations are red and yellow, and current is red. All values in 3.6b have been normalized to 1.

ent from simple redox reactions. Recently, Zu et. al. performed a series of PFV experiments [82], in which they adsorbed the Iλ subunit of Complex I onto a protein film electrode and measured a voltammogram for the reversible overall redox reaction of this subunit, 3.3. This voltammogram is presented in fig. 3.7. The voltammogram in fig.3.7 exhibits behavior which is explainable by 3.27 at the low potential limits, but high potential limits appear to tend towards two different values rather than a a single mass limited current which is the result of a mass transport limited system such as 3.29. Furthermore, as stated in [82], the electrode was continuously rotated as to minimize the effects of any depletion layer. Thus, the limiting currents appear to a result of a limiting transfer of the internal enzyme catalysis, or a limiting behavior of the interfacial electron transport.

Even though the overall reaction for NADH oxidoreductase 3.3 is known, the mechanism for this reaction is currently an active topic of research. A wide range of models currently exist to model such electro-catalytic reactions. Léger et. al. developed a model Potential-

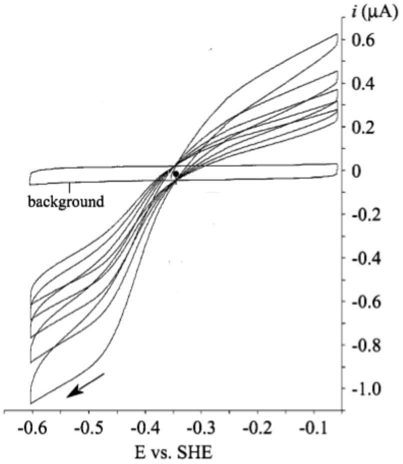


Figure 3.7: Experimental cyclic voltammogram of Iλ subunit of Complex I. Taken from [82].

Dependent Michaelis-Menten Model which has a limited number of free parameters and is potential and substrate concentration dependent [51]. This model assumes fast mass transport and interfacial electron transport. Léger et. al. also proposed another model which provide more physical insight for certain CV wave forms [52]. This model also uses a Butler-Volmer type term such as the one developed here. Sucheta et. al. developed a widely used model which treats the interfacial electron transfer, enzyme catalysis and mass transport as a set of resistors in series, each with a limiting current based on Michaelis-Menten kinetics [74]. Heering et. al. also developed a set of models based on a Butler-Volmer for the interfacial electron transport and a Michelis-Menten approach for including a limiting current [39]. Reda et. al. developed a very sophisticated model model which offers significant physical insight, however has a large number of free parameter [63].

The mechanism may not be understood well, but we can still model the reaction based on experimental data and the known behaviors at upper and lower limits. We know that at low limits, the reactions must be concentration and potential limited. Even though the exact mechanism by which the Iλ reaction tends to different current limits in the upper and lower potential limits is unclear, the experimental data in fig. 3.7 clearly indicates there there are different current limits. The current limits also do not appear to agree with the Levich equation [6], however these limits may be inferred from the experimental data. Therefore, instead of uniform limiting current, we model the limiting current as a function of the electrode potential which tends to different values for the upper and lower limits as

$$i_{lim} = \frac{i_r + i_f \exp(\phi + E^\circ)}{1 + \exp(\phi + E^\circ)}, \tag{3.30}$$

where i_f and i_r are the experimentally observed forward and reverse limiting current values, and ϕ is the electrode potential. In a simple redox reaction, rate of current change per potential in the high and low potential limits is symmetric about E° and is determined by the single α parameter. In the Iλ subunit reaction, the experimental data clearly indicates that these slopes are not symmetric. In order to account for this behavior, the α parameter may be split into two components, α which effect the slope of the forward or oxidation reaction, and β which only effects the slope of the reverse reaction. With the separated α and β parameters, the forward and reverse rates become

$$\tilde{R}^+ = R^+ \exp\left(\alpha \frac{nF}{RT}(\phi + E^\circ)\right) \;;\; \tilde{R}^- = R^- \exp\left(-\beta \frac{nF}{RT}(\phi + E^\circ)\right), \tag{3.31}$$

where again, R^+ and R^- are the forward and reverse mass-action rates. Finally, in order to tie in the low potential and high potential limits, we will use a Michelis-Menten approach, this will introduce a Michelis-Menten coefficient, k_m which determines where fast the low and high potential limits intersect. Thus, the net rate of reaction becomes

$$\begin{aligned}\tilde{R} &= k^\circ \left(R^+ \exp\left(\alpha \frac{nF}{RT}(\phi + E^\circ)\right) - R^- \exp\left(-\beta \frac{nF}{RT}(\phi + E^\circ)\right)\right) \\ R_{net} &= \frac{i_r + i_f \exp(\phi + E^\circ)}{1 + \exp(\phi + E^\circ)} \frac{\tilde{R}}{k_m + |\tilde{R}|}.\end{aligned} \tag{3.32}$$

In order to estimate values for the Michelis-Menten coefficient, and the forward and reverse coefficients, the data from [82] was digitized and the parameters were fitted. This resulted in parameter values of

$$\alpha = 0.07,\ \beta = 0.16,\ k^\circ = 3.16 \times 10^{-6},\ i_f = 0.5,\ i_r = 1.25,\ E^\circ = 0.34. \tag{3.33}$$

The current using the reaction rate specified in 3.32, with the above parameters is plotted

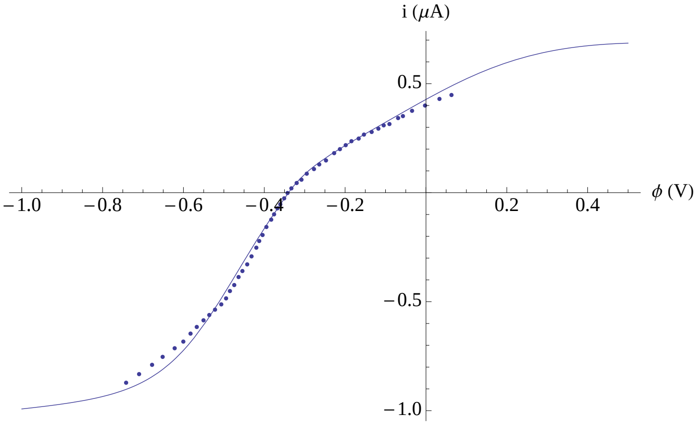


Figure 3.8: Digitized experimental data from 3.7 and the electric current resulting from the rate 3.32 evaluated with the parameters 3.33.

against the digitized data from [82] in fig. 3.8.

These results capture both the how and low electrode potential behavior of experimental data with a minimal number of free parameters which are easily determined from readily available experimental data. As with any empirical model, there is a danger of over-fitting with too many parameters. The model developed here has minimal number of free parameters. Although the models developed in [63] more accurately fit experimental data, they contain a large number (~ 20) of free parameters.

Even though the model developed here does not attempt to accurately describe the internal electro-catalytic mechanisms of subcomplex Iλ, it does accurately correspond with experimental data and is generalizable to a variety of electro-catalytic reactions. Empirical models are commonly used in many areas of chemistry such as molecular dynamics. Here, no attempt is made to accurately describe the fundamental quantum mechanical nature of chemical compounds, rather a series of Hookean spring forces are fitted to experimental data and this is used to provide numerical solutions to a wide range of biophysical phenomena.

3.4 Spatial Electro-Chemical Kinetics

3.4.1 Electrical Double Layer

The presence of the dielectric decreases the electric field produced by a given charge density. The laws of electrostatics in a polarizable medium are exactly the same as in vacuum, permittivity of free space, ϵ_0 is replaced by the material permittivity, ϵ. The effective electric field is always smaller in a dielectric medium as in vacuum.

Most eukaryotic cells maintain an out of equilibrium resting potential of -40 to -80 mV, with the cell interior typically being negative.

The number density of changed ions falls away exponentially from a charged membrane. Beyond the Debye length, the potential is essentially zero and the charge distribution is uniform. The charged ions are distributed in a thin layer of approximately the Debye length in thickness. Under typical biological conditions, the Debye length is 0.7 nm [65]. All excess ionic charge will build up in this thin layer.

3.4.2 Physical Description

In a cyclic voltammetry experiment, the electrode face is very large relative to the diffusion length, the electrode face is on the order of $1\,\text{cm}^2$. Therefore, it is reasonable to model the system as a one dimensional problem where we take advantage of the uniformity of the y and z directions, and only consider the x direction.

The regions between the electrode face and the bulk solution can be treated as a series of fictitious compartments. These compartments can be thought of as regions of space in a finite element simulation. Unlike traditional multi-compartment simulations, there is no physical boundary in this case, each compartment is in direct contact with the neighboring compartments.

The left most compartment is treated as the *bulk* compartment. This region represents the bulk solution and all species here are considered boundary species in that their values are constant. The rightmost compartment is in direct contact with the electrode face. This compartment is treated as the *debye* compartment. The electrical field produced by the electrode will attract a significant amount of charged solutes to this compartment and the majority of charge screening will occur here. The center compartment is treated as the *reaction* compartment. Most of chemical reactions will occur here such as the oxidation / reduction of NADH NAD+. The NAD oxidoreductase protein is anchored to the electrode face, however the protein is rather long and extends into this region, therefore, the reaction will occur here. The reduction potential will be the difference between the center of the reaction compartment and the electrode face. We can now isolate a single compartment as in 3.9.

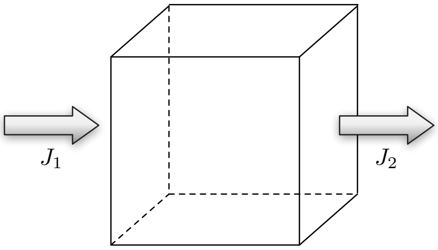


Figure 3.9: Flux of a substance through cube

Conservation of mass dictates that the the net accumulation of mass is equal to the sum of the fluxes and the sum any local sources and sinks:

$$\text{mass accumulation} = \text{flux in} - \text{flux out} + \text{source} - \text{sink}$$

The flux of a substance in a particular direction is defined as the amount of that substance passing through a surface perpendicular to the velocity per unit time 3.10. Flux is

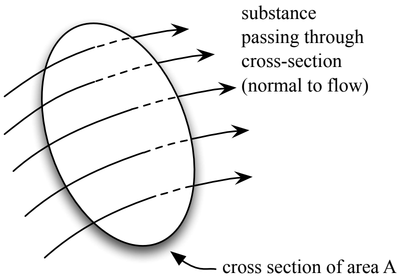


Figure 3.10: Flux of a substance through a surface

defined as rate of flow of a substance per unit area, and has dimensions of $^{\text{mass}}/_{\text{area}\times\text{time}}$. The term *flux* can also also refer the rate of change of amount of a substance participating in a reaction. The SI unit of flux is $^{\text{mol}}/_{\text{m}^2\text{s}}$. On inspection of 3.9, we can see that the amount of substance entering the left side is $J_1(x)A$, where A is the area of a cube face, and x is the position of the left face. Similarly, the amount leaving the right side is $J_2(x+\delta x)A$. Any difference in the fluxes could only have come from, been deposited into the cube. Here we have made the assumptions that the flux is uniform in the y an z directions. If we make the restriction that the flux must be a smooth, differentiable quantity in all directions, than it is always possible to choose a cube of volume sufficiently small to ensure that the flux is uniform in the perpendicular directions. We also make the assumption that geometry of the region does not change during the duration δt. The change in the amount of a substance in the cube per unit time δt is then $V{}^{\delta c}/_{\delta t}$, where V is the volume of the cube, c is the concentration, and t is time. Taking the limit as δt and δc tend to zero, we may write the rate of change of concentration in a region as

$$\frac{dc}{dt} = -\frac{A}{V}J. \tag{3.34}$$

Here, the convention is that the magnitude of the flux vector is the flux vector dotted into the surface normal vector of the cube face. The surface normals point outwards, a positive change in concentration will result from a positive flux flowing inwards.

The total rate of change of a substance in a compartment is the sum of fluxes of that substance. Fluxes can arise from passive transport, the electro-diffusive flux, active reactions in the compartment and in the case of compartments separated by physical barriers, from active transport. In the case of determining the electrical double layer potential, we will first only consider passive diffusive flux.

The passive flux resulting from a concentration differential between compartments is defined via Ficks law as

$$J_f = -D\frac{dc}{dx}, \tag{3.35}$$

where D is the diffusion coefficient, in units of ${}^{\text{length}^2}/_{\text{time}}$. Fick's law can be derived in a variety of ways, but here, we will quickly review a derivation based on Brownian motion. Returning to 3.9, but now consider two volumes on each side of a boundary plane. We can define the average distance that a particle may travel per unit time τ as $\delta = v\tau$, where v is the average particle velocity. Now, we define the size of each volume to be length δ on each side, and a contact area of A. We make the assumption that δ is small enough to assume an average concentration of $c(t, x{-}\delta/2)$ in the left volume, $c(t, x{+}\delta/2)$ to the right. The average number of molecules in the left and right volumes are $\delta Ac(t, x - \delta/2)$ and $\delta Ac(t, x + \delta/2)$

respectively. From the random walk model, half the particles in each volume would cross over into the other volume during a τ time span. So, the average number of particles being exchanged in each volume during time τ is $1/2\delta Ac(t, x - \delta/2) - 1/2\delta Ac(t, x + \delta/2)$. The flux $J(x, t)$ between volumes then becomes

$$J(t,x) = -\frac{1}{\tau A}\left(\frac{\delta Ac(t,x-\delta/2)}{2} - \frac{\delta Ac(t,x+\delta/2)}{2}\right) \tag{3.36}$$

$$= -\frac{\delta^2}{2\tau}\left(\frac{\delta Ac(t,x-\delta/2) - \delta Ac(t,x+\delta/2)}{\delta}\right). \tag{3.37}$$

Taking the limit as $\delta \to 0$, we arrive at Fick's first law, 3.35, where the diffusion coefficient is defined as $D = {}^{\delta^2}/_{2\tau}$.

The active components of the flux are driven by potential energy differences between regions. In the diffusion dominated, low Reynolds number regime of our system, in the presence of a potential energy gradient, the motion of the solute particles are still limited by random collisions, such that the velocity, rather than the acceleration is proportional to any applied force. In such regimes, *drift velocity* or average velocity of solute particles is given by

$$v(x,t) = \mu F(x,t) \tag{3.38}$$

$$= \mu \nabla U(x,t). \tag{3.39}$$

Here, $v(x, t)$ is the drift velocity, μ is the mobility, and F and U are the applied force and potential energy gradients. Returning again to 3.9, the geometry gives us that the molar flux at any moment in time is given by the molar concentration measured at the face of the cube times the drift velocity:

$$J(x,t) = c(x,t)v_d(x,t) \tag{3.40}$$

$$= \mu c(x,t)F(x,t) \tag{3.41}$$

$$= \mu c(x,t)\nabla U(x,t). \tag{3.42}$$

The mobility, μ given in ${}^{\mathrm{m^2 mol}}/_{\mathrm{Js}}$ relates the applied force to the resulting drift velocity, and is given by the Einstein relation as

$$\mu = \frac{D}{K_b T}. \tag{3.43}$$

The flux of charged species due to electrophoretic effects can be defined in terms of the electrical potential *phi*, concentration c and ion mobility u as

$$J_e = -uc\frac{d\phi}{dx}. \tag{3.44}$$

The net passive flux is the sum of 3.35 and 3.44 as

$$J_p = -D\frac{dc}{dx} - uc\frac{d\phi}{dx}. \tag{3.45}$$

The diffusion coefficient is typically obtained experimentally, however, using the Nernst-Einstein relation, it may be defined in terms of the mobilities as

$$D = \frac{u}{z}\frac{RT}{F}, \tag{3.46}$$

arriving at the Nernst-Planck electro-diffusion equation

$$J_{np} = -D\left(\frac{dc}{dx} + c\frac{zF}{RT}\frac{d\phi}{dx}\right). \tag{3.47}$$

The simplest physical described by the Nernst-Planck equation is depicted in Fig. 3.11. Here, two uniform regions of space are separated by a fictitious boundary. The system contains spatially invariant concentrations of Na$^+$ and Cl$^-$. Each compartment is held at electrical potential, ϕ_1 and ϕ_2 respectively. The fluxes between "compartments" can be determined by a naïve finite-difference approximation of the NP equation as

$$J_{12} \approx -D\left(\frac{c_2 - c_1}{\delta x} + c_1\frac{zF}{RT}\frac{\phi_2 - \phi_1}{\delta x}\right) \tag{3.48}$$

$$J_{21} \approx -D\left(\frac{c_1 - c_2}{\delta x} + c_2\frac{zF}{RT}\frac{\phi_1 - \phi_2}{\delta x}\right). \tag{3.49}$$

And the net flux between compartments is the sum of J_{12} and J_{21}. Note, that flux behaves

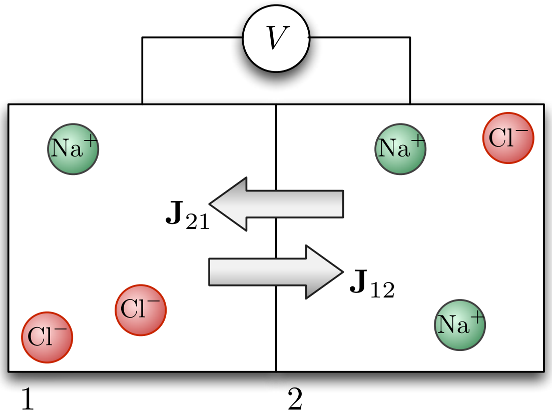


Figure 3.11: Flux between two fictitious compartments

as a vectorial quantity in that it has a direction. The flux in the forward (left to right) direction is written as

$$J = J_{12} + (-J_{21}) \tag{3.50}$$

$$= \frac{D}{\delta x}\left(2\left(c_1 - c_2\right) + \frac{Fz\left(c_1 + c_2\right)\left(\phi_1 - \phi_2\right)}{RT}\right). \tag{3.51}$$

Here, Eqn. 3.51 describes the rate of mass transfer per unit time per unit area between compartments 1 and 2. In general, each compartment may have different volumes. Taking the both the contact area between compartments, as A, and their respective volumes into account, we arrive at a pair of equations describing the rate of change of concentration in each compartment due to the electro-diffusive flux between them:

$$\frac{dc_1}{dt} = \frac{A}{V_1}J, \tag{3.52}$$

$$\frac{dc_2}{dt} = -\frac{A}{V_2}J. \tag{3.53}$$

This form suggests that all trans-compartment fluxes may be written as chemical reactions. Any electro-diffusive (or any other flux for that mater) may be written as a chemical reaction which consumes a substance in one compartment and creates an equivalent amount of that substance in a neighboring compartment as

$$\mathrm{C}_1 \xrightleftharpoons{AJ} \mathrm{C}_2 \tag{3.54}$$

where the reaction rate is defined as AJ. Note that a chemical reaction rate has units of $^{\text{mass}}/_{\text{time}}$. The flux is defined as $^{\text{mass}}/_{\text{area}\times\text{time}}$, thus must be scaled by the interfacial area in order to correctly contribute to a reaction. This has the physical interpretation that as the interfacial area tends to zero, the exchange of mass between compartments also tends to zero.

For small potentials, the finite difference approximation is valid, however, one may notice a key problem in that it allows the forward flux out of a compartment with zero concentration. This will drive the concentration negative which is unphysical.

We make a few physical approximations that allow us to write a closed form of the compartment flux. Following same line of reasoning as the Goldman-Hodjkin-Kats equation, we make derive a flux relation between a pair of adjoining spatial regions.

We have already assumed a planar geometry, as in both the experimental voltammetry system, and the biological mitochondrial membranes are significantly larger than the length scale of the processes occurring at the membrane face. The x axis is normal to the electrode plane, and we define a length l which is the center to center distance of each compartment. This is the average length that a particle moves between a pair of compartments. We now make the approximation that the average force that a particle feels as it moves from one compartment to another is constant. Therefore, we may write the average force as the difference in potential energies between each region,

$$F = \frac{dU}{dx} \approx \frac{U(l) - U(0)}{l} = \frac{U_l}{l}. \tag{3.55}$$

With 3.55 the flux equation 3.56 then becomes

$$J_k = -D_k \left(\frac{dc_k}{dx} + \frac{c_k}{K_b T} U_l \right), \tag{3.56}$$

for the k^{th} species. This now becomes an analytically solvable ODE. Re-arranging 3.56 to an integrable form, we have,

$$dx = \frac{dc_k}{-\frac{J_k}{D_k} - \frac{c_k U_l}{K_b T}}. \tag{3.57}$$

Integrating from the center of the source compartment on the left ($x = 0$) to the center of the destination compartment ($x = l$). Here, we assume ...

$$\int_0^l dx = l = -\frac{K_bT}{U_l} \ln \left(\frac{\frac{U_l}{K_bT} c_k^l + \frac{J_k}{D}}{\frac{U_l}{K_bT} c_k^0 + \frac{J_k}{D}} \right), \tag{3.58}$$

where c_k^l is the concentration at $x = l$, the center of the destination compartment, and c_k^0 is the concentration at the center of the source compartment, $x = 0$. Solving for J_k, we arrive at

$$J_k = -D \frac{U_l}{K_bT} \frac{c_k^l - c_k^0 \exp\left(-U_l/K_bT\right)}{1 - \exp\left(-U_l/K_bT\right)}. \tag{3.59}$$

Numerically, eqn. 3.59 still has one issue, that being when the potential energy difference tends to zero. Using l'Hôpital's rule, we make take the limit of $U_l \to 0$ of 3.59 and as one would expect, we recover the finite difference approximation of Fick's flux law as

$$J_k = -\frac{D_k}{l} \left(c_k^l - c_k^0 \right). \tag{3.60}$$

As an implementation detail, we check if U_l is near machine ϵ, and if so, return 3.60, otherwise, 3.59. Accounting for the zero potential condition, the GHK flux is written as

$$ghk(D, z, c_1, c_2, z, \phi, l) = \begin{cases} -\frac{D_k}{l} \frac{zF\phi}{K_bT} \frac{c_1 - c_2 \exp\left(-zF\phi/K_bT\right)}{1 - \exp\left(zF\phi/K_bT\right)} & |\phi| <= \epsilon \\ -\frac{D_k}{l} \left(c_1 - c_2\right) & |\phi| > \epsilon. \end{cases} \tag{3.61}$$

One of the key advantages of this formalism is that all processes influencing the dynamics of species may be written as chemical reactions. This allows a modeler to first start with a basic set of processes and refine the model over time as further analysis and experimental data become available. All future processes would simply be added as another reaction without having to re-derive or fundamentally alter the basic model.

3.4.3 Non-Dimensionalization

From a numerical perspective, equations 3.51, 3.53 are often ill-conditioned in their present form. The flux must be scaled by the surface area to obtain the correct physical units, and the numerical values of each side are often vastly different. Even though the equations discussed here are are intended to describe a specific cyclic voltammetry simulation, their form is completely general and the physics behind them is universal in any classical regime.

When using the physical constants appropriate for a cyclic voltammetry simulation, the species amounts tend to 10^{-14} mol as the compartment volumes are in the 10^{-11} liter range. The use of the present physical constants pose serious issues for numerical solution of these equations as as the state variables are proportional to *machine* ϵ.

When solving any set of equations by numerical means, one has be very aware of issues such as machine precision and round off error. Numbers are typically stored in software as single or double precision floating point values [1]. This specific number of bits implies that floating point representations have finite precision and finite range. Floating point numbers have a discrete spacing and minimum and maximum values. Machine epsilon (ϵ) is defined to be the smallest positive number which, when added to 1, gives a number different from 1. This, in effect is the smallest difference between numbers that can be differentiated. ϵ is typically approximately 10^{-16} for double precision numbers. Therefore, using double precision, $(1+1\times10^{-16})-1 == 0$, but $(1+2\times10^{-16})-1 == 2.2204\times10^{-16}$. When numbers approach the range of ϵ, the round off error becomes proportional the numbers themselves. Therefore machine arithmetic can not be practically performed with the numbers are in this range.

When implementing these equations in software, at one point or another, we have to work with numbers and at this point the physical units are lost. If one does not take care at this point, it is easy to get software in which all accuracy is lost due to round off errors. On the other hand, non-dimensionalization typically avoids this since it normalizes all quantities so that values that appear in computations are typically on the order of one.

From an algebraic perspective, it is typically clear how to non-dimensionalize an equation with constant coefficient. However, if the dynamics values or coefficient vary by several orders of magnitude, one has to choose a reference parameters such as volume and amount in order to normalize the equations.

On the downside, the numbers non-dimensionalized equations produce are not immedi-

---

[1] Numbers may also be stored and operated on using arbitrary precision arithmetic, however it is relatively rare that numeric simulations are performed using arbitrary precision. Arbitrary precision arithmetic is typically orders of magnitude slower than conventional floating point arithmetic.

ately comparable to ones we know from physical experiments. This is of little concern if all we have to do is convert every output number of our program back to physical units. On the other hand, it is more difficult and a potential source of errors.

3.5 Electrical Double Layer

3.5.1 Electrical Double Layer Models

The first model of electrolyte charge separation at an interface boundary was proposed by Helmholtz in 1853 [38]. This simple model proposes that interfacial area between a conducting electrode and electrolyte solution behaves as a parallel plate capacitor, thus has the ability to store charge. This model states that an electrode has a charge density q which is the result of an surplus or deficit of surface electrons. Any available charge in the electrolyte solution will exactly balance the electrode surface charge. All charge is assumed to rigidly held in a thin parallel surface on the electrode.

The volume directly outside the facial area of a voltammetry electrode can be approximated as set of three compartments. Each compartments here is just a physical location of space, there is no membrane or other barrier separating them. All solutes are free to diffuse between neighboring compartments.

We will consider the bulk region as having zero potential. In actuality, the average potential of the bulk compartment is $V_e/2$, with V_e being the electrode potential.

The Debye compartment is a relatively thin compartment directly facing the electrode. The majority of the charge screening layer will accumulate here, and in fact, a single compartment is exactly the Helmholtz double layer model describes [6].

The reaction compartment is where all of the reactions will occur. NAD Oxidoreductase is large trans-membrane protein which in vivo is embedded into the inner mitochondrial membrane [53]. In a voltammetry experiment, the matrix facing alpha subunit is cleaved off and attached to an artificial voltammetry protein electrode. The electrode takes the place of the mitochondrial double plasma membrane, and acts as an artificial charge carrier substituting for the Quinones. The alpha subunit is itself a large protein, and extends into the mitochondrial matrix well beyond the Debye layer. This protein experiences a potential difference of the electrode potential and the reaction compartment potential, which will effect the electro-chemical potential determining whether it will catalyze the oxidation or reduction of NAD or NAD+ respectively.

3.5.2 Electric field from the charged compartments

As the surface area of the electrode is significantly larger that the thickness of the compartments, we can approximate each compartment as a uniformly charge, infinite extent slab. The charge per unit volume in the slab is ρ. We may draw a perpendicular Gaussian cylinder enclosing the slab, with each face parallel to the slab face. The total electric flux is $\Phi_E = 2|E_z|A$. We will only be concerned with the potential at the center of each compartment, and at the exact center, the electric field produced by the compartment is zero (the charge on each side of the centerline cancel each other). Thus we will only be concerned with electric field produced by the compartment outside the compartment. Here, the total charge enclosed is $Q_{enc} = \rho A d$, where A is the surface area of the Gaussian cylinder, and d is the thickness of the compartment. The net electric field is then

$$E_z = \begin{cases} -\frac{\rho d}{2\epsilon} & x < 0 \\ \frac{\rho d}{2\epsilon} & x > 0 \end{cases} \tag{3.62}$$

For each electrode facing compartment, the compartment thickness terms, d cancels with the volumetric charge density, $\rho = q/v$, yielding the net electric field produced by each compartments as

$$E_z = \begin{cases} -\frac{q}{2A\epsilon} & x < 0 \\ \frac{q}{2A\epsilon} & x > 0, \end{cases} \tag{3.63}$$

where A is the electrode facial area. Note, this is the same electric field produce by an infinite sheet of charge, where $\sigma = q/A$.

Each region or compartment can be approximated as a slab of finite thickness, but very large area relative to thickness. The concentration of all electrolytes within each compartment is treated as constant. Application of Gauss's law to a finite thickness slab

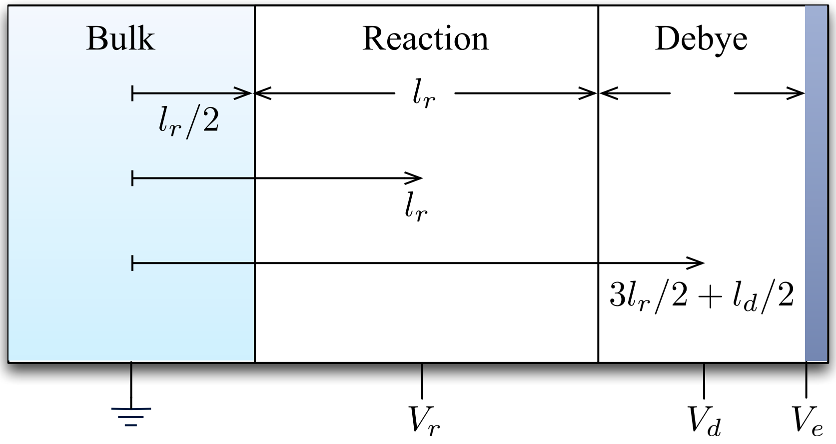


Figure 3.12: Physical layout of compartments. The region is divided into three compartments: bulk, reaction and Debye. The reaction and Debye have lengths of l_r and l_d respectivly, and by convention, the diffusion length from the bulk to reaction is l_r. By convention, the bulk compartment is grounded, and the reaction and Debye compartments have their potentials, V_r and V_d measured at the center of the compartments. The electrode has a potential of V_e.

yields an electrical potential of

$$\begin{cases} -\frac{q}{2A\epsilon} & x < -l/2 \\ \frac{qx}{A\epsilon l} & l/2 \leq x < l/2 \\ \frac{q}{2A\epsilon} & x \geq l/2, \end{cases} \tag{3.64}$$

where l is the thickness of the compartment, A is the area of the compartment, q is the net charge, and x is the spatial coordinate. Integrating with respecting to the spatial coordinate, x, and choosing the electrical potential of the center section evaluated at the endpoints as the constant of integration of outer sections, we arrive at the electrical potential for a uniformly charged slab:

$$\begin{cases} -\frac{lq}{8A\epsilon} + \frac{q\left(\frac{l}{2}+x\right)}{2A\epsilon} & x < -l/2 \\ -\frac{qx^2}{2A\epsilon l} & l/2 \leq x < l/2 \\ -\frac{lq}{8A\epsilon} - \frac{q\left(-\frac{l}{2}+x\right)}{2A\epsilon} & x \geq l/2. \end{cases} \tag{3.65}$$

The electrical potential and field for a uniformly charged slab are shown in 3.13.

The electrical potential is measured at the center of each compartment. The center of

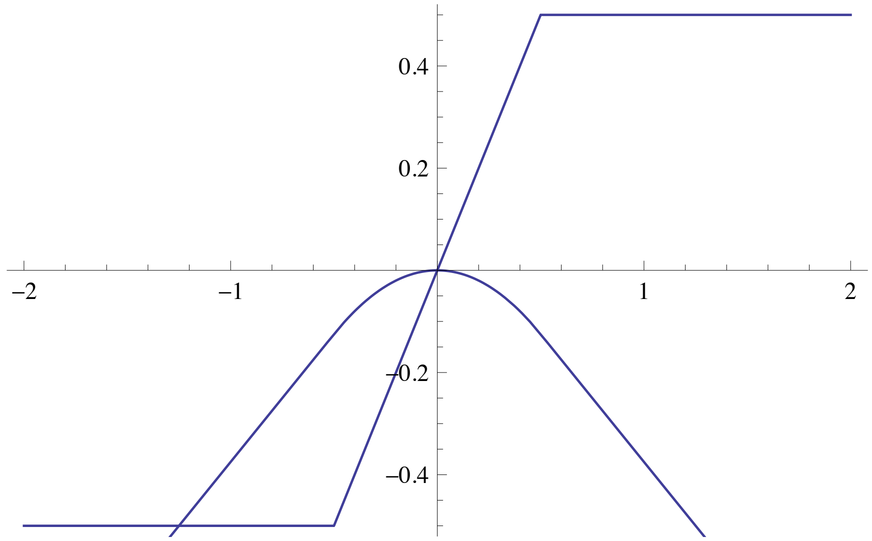


Figure 3.13: Electrical field and potential for uniformly charged slab

the reaction compartment in our coordinate system is located at $x = l_r$, and the center of the Debye compartment is located at $x = {}^3\!/_2 l_r + {}^1\!/_2 l_d$. The potential due to the reaction compartment is obtained by shifting the slab potential 3.65 as:

$$V_r(x) = \begin{cases} \frac{\left(x - \frac{l_r}{2}\right) q_r}{2A\epsilon} - \frac{l_r q_r}{8A\epsilon} & x - l_r < -\frac{l_r}{2} \\ -\frac{(x-l_r)^2 q_r}{2A\epsilon l_r} & -\frac{l_r}{2} \leq x - l_r < \frac{l_r}{2} \\ -\frac{\left(x - \frac{3l_r}{2}\right) q_r}{2A\epsilon} - \frac{l_r q_r}{8A\epsilon} & x - l_r \geq \frac{l_r}{2}. \end{cases} \tag{3.66}$$

Similarly, the potential due to the Debye compartment becomes,

$$V_d(x) = \begin{cases} -\frac{(-4x + l_d + 6l_r) q_d}{8A\epsilon} & 2x < 3l_r \\ -\frac{(-2x + l_d + 3l_r)^2 q_d}{8A\epsilon l_d} & 3l_r \leq 2x < 2l_d + 3l_r \\ \frac{(-4x + 3l_d + 6l_r) q_d}{8A\epsilon} & 2x \geq 2l_d + 3l_r. \end{cases} \tag{3.67}$$

It is assumed that the potential of the bulk compartment is that of the reference electrode, which by convention is zero. In the absence of any electrolyte, the potential of the between the bulk and the working electrode is a linear function which rises from zero to working electrode potential, $V_e(t)$. The potential due to the electrodes is refered to as V_0 and is given by

$$V_o = \frac{x}{L} V_e. \tag{3.68}$$

Using the principle of linear superposition, the net electrical potential due to the charged layers and the elctrode is the sum of each component, this is given by

$$V_{net}(x) = V_r(x) + V_d(x) + V_0(x). \tag{3.69}$$

The molar flux between each region is given by 3.59, this flux equation depends on the electrical potential differences between each reagion. Therefore, the net electrical potential, 3.69 is evaluated at the center of the Debye and bulk compartments, these potentials are given by

$$V_d = V_{net}(x_d) = \frac{-\left(4l_d^2 + 8l_d l_r + 3l_r^2\right) Q_r + 8Ae(l_d + 3l_r)V_e}{8Ae(2l_d + 3l_r)} \tag{3.70}$$

$$V_r = V_{net}(x_r) = -\frac{2l_d^2 Q_d + 7l_d l_r Q_d + 6l_r^2 Q_d - 16Ael_r V_e}{16Ael_d + 24Ael_r}. \tag{3.71}$$

Here, Q_r and Q_d are the total charge in the reaction and Debye compartments, A is the interfacial area of the compartments, and e is the elementary charge.

A model was constructed which contains Na and Cl in all compartments.The values of these reactants were fixed in the bulk compartment, and a constant electrical potential was applied to the electrode, and the potential in the bulk was fixed at zero. The only reactions in the system were the diffusion reactions between each compartment, these are given by

$$\begin{aligned}
\mathrm{Na_r} \longleftrightarrow \mathrm{Na_d} &\ :\ ghk(D_{Na}, 1, [Na_r], [Na_d], V_d - V_r, l_{rd}) \\
\mathrm{Na_b} \longleftrightarrow \mathrm{Na_r} &\ :\ ghk(D_{Na}, 1, [Na_b], [Na_r], V_r, l_{br}) \\
\mathrm{Cl_r} \longleftrightarrow \mathrm{Cl_d} &\ :\ ghk(D_{Cl}, -1, [Cl_r], [Cl_d], V_d - V_r, l_{rd}) \\
\mathrm{Cl_b} \longleftrightarrow \mathrm{Cl_r} &\ :\ ghk(D_{Cl}, -1, [Cl_b], [Cl_r\, V_r, l_{br}).
\end{aligned} \tag{3.72}$$

3.5.3 Double Layer Capacitance Results

The capacitance of the electrical double layer model was measured whilst it was being driven by a slowly varying potential (1Hz). Such a slow frequency was chosen so that the charge distribution may equilibrate with the boundary conditions. This slow change in the boundary conditions characterized an adiabatic process. In this regime, the capacitance vs potential curves closely match that of the Stern double layer model. Note, the Stern (and all of the classical models are only valid at steady state).

Using the Gouy-Chapman-Stern (GCS) approximation, the differential capacitance of the Debye layer is given b [6]

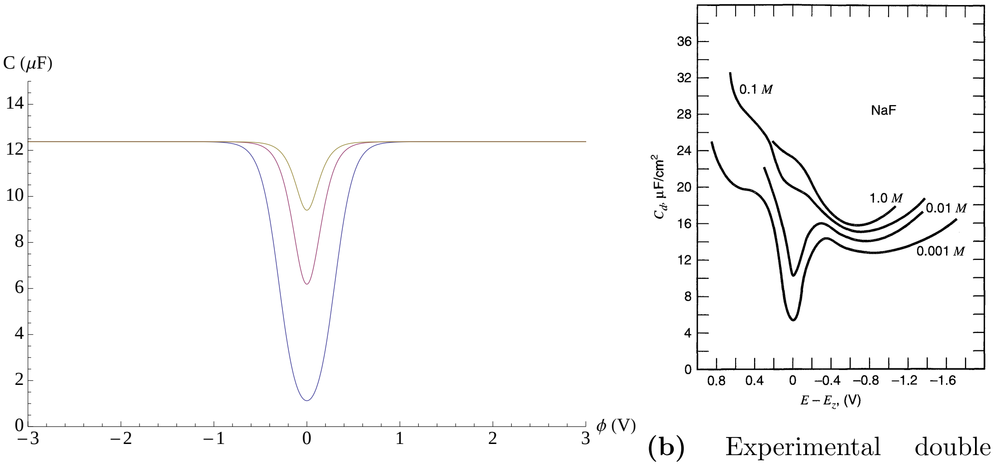
$$C_d = \frac{\left(2\epsilon\epsilon_0 z^2 e^2 n^0\right)^{1/2} \cosh\left(ze\phi/2K_bT\right)}{1 + \left(x_2/\epsilon\epsilon_0\right)\left(2\epsilon\epsilon_0 z^2 e^2 n^0\right)^{1/2} \cosh\left(ze\phi/2K_bT\right)}, \tag{3.73}$$

where e is the elementary charge, n^0 is the bulk electrolyte concentration, z is the valence number, ϕ is the applied potential, and x_2 is the Stern layer thickness. For a detailed derivation, see [6].

The GCS approximation assumes a double layer model where the inner Stern layer (closest to the electrode face) has a tightly packed but limited ion concentration, and the longer diffuse layer has an exponential decrease of ion concentration with the inner side having the same concentation as the Stern layer, and the outer side has the same concentation as the bulk medium. This models shows a quadratic behavior of the capacitance for low potentials and approaches a fixed value for high potentials. The Gouy-Chapman (GC) model alone assumes only a diffuse layer. This model shows correct behavior for low electrical potentials, however as there is no limiting condition for large potentails, the ion concentration in the diffuse layer will tend to infinity at large potentials. This results in the capacitance also tending towards infinity, which is un-physical. A plot of the GCS model is shown in fig. 3.14a.

Experimentally, the double layer capacitance shows low potential behavior similar to both the GC and GCS models, and is of course limited at high potentials [35]. However, unlike the GCS model, experimental results show that the capacitance at larger potentials does not have the same limit for large positive and negative potentials. Also, there is a slight rise in capacitance as the low and high potential limits connect. Presently, a simple anallytic model in the style of the GCS model which reproduces the different capacitance limits does not appear to exist [6]. An experimental measurments of the double layer capacitance of an Na-F solution is presented in fig. 3.14b.

The key behavior that both the CGS and experimental results show is that the capacitance is strongly depedent on the bulk electrolyte concentration. A low bulk concentration will show a drop in the capacitance for low potentials as the bulk concentration is reduced. For high potentials, a strong electrical field is sufficient to overcome the natural diffusive tendencies of the electrolytes and establish high ion concentraton Stern and diffuse layers which results in the same high potential limit capacitance.



(a) Analytic solution using the GCF approximation

(b) Experimental double layer capacitance from [35]

Figure 3.14: Analytic and experimental double layer capacitance results. Both show a strong depencence of the double layer capacitance on the bulk ion concentration. The anallytic model is a plot of 3.73 with the bulk ion concentrations of 0.01, 0.5 and 1 M.

In order the test the correctness of the multi-compartment model developed here, the double layer capacitance was measured and compared to analytic and experimental results. A model was constructed using the four reactions in 3.72, the boundary concentration of Na and Cl was set to fixed values and the system was run to a steady state using the built in steady state solver. The capacitance was measured via rule which summed to the total charge in the Debye and reaction compartments and divided this by the applied electrode voltage. These resulting C-V curves are displayed in fig. 3.15.

The results show that the multi-compartment model developed here has qualitative agreement with the the GCS model for low concentrations in the both the low and high potential limits. However, for higher concentrations, the multi-compartment model exhibits a slight rise in capacitance for low potentials before reaching the same high potential limit when the electrolyte concentration is high. This does have the same qualitative behavior at least in this respect as the experimental results. While this rise in capacitance is qualitativly similar to experimental results, the physical origin of this behavior is thoought to arise from the mutual electrostatic repulsion of ions in the double layer in directions parallel to the interface [35]. The model developed here does take this into account, and the behavior exhibited here is likely an artifact of coarse numerical descritization (large compertment size).

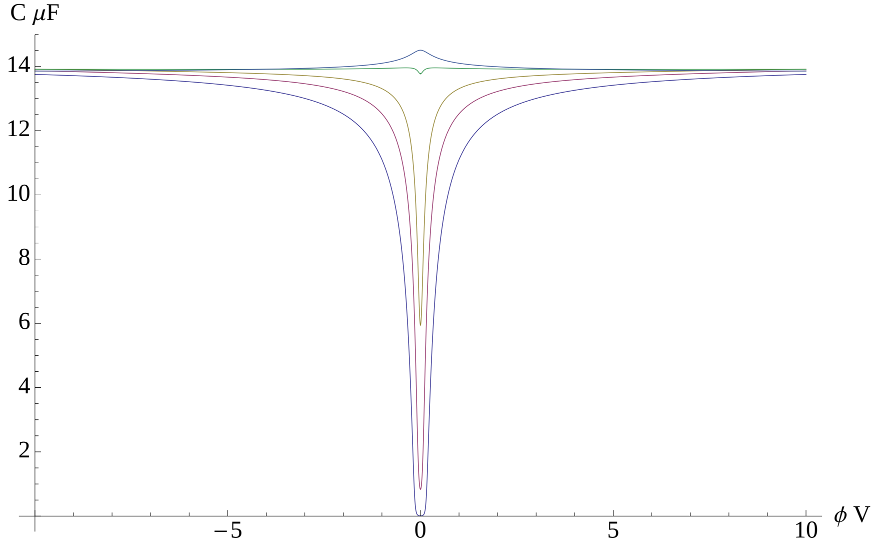


Figure 3.15: capacitance vs. voltage at different molarities of NaCl solution. The bulk concentation of Na and Cl was fixed at 0.00001, 0.0001, 0.001, 0.01, 0.07 M

3.6 Conclusions

Surface redox reactions almost always have asymmetric limiting currents – the oxidation and reduction limiting are typically different values. Electro-catalytic proteins have highly complex internal mechanisms and a simple symmetric limiting current does not account for the observed behavior. PFV experiments typically use a stirred electrode which is intended to reduce or eliminate the depletion affects. Yet, PFV experiments show limiting currents even in the absence of depletion affects. In order to account for this complex behavior, a relatively simple model which correctly explains most biological redox reactions with minimal number of parameters was developed. This model has parameters have physical meaning and easily obtained from common experiments. Using the approach developed here, it is possible to develop relatively simple models of electro-catalytic proteins which are easily calibrated with experimental data and are specifiable in a re-usable format.

Chapter 4

Stochastic Bi-Stability Analysis

4.1 Introduction

Traditionally, there have been two methodologies for mathematically analysing a given system: the deterministic, and the stochastic methodology. The deterministic view treats a system as a completely predictable set of dynamics, described by a set of ordinary or partial differential equations. The central tenant of the deterministic view is that if we know with absolute certainly the current state of the system, the we can predict, for all time, all future states of the system.

The stochastic viewpoint on the other hand deals with probabilities, not certainties: if we have a measurement of the system at the current state, and we know the probability rate equation that govern the system, we can predict the probability distribution of what states the system will likely be in in the future.

The deterministic viewpoint has always been simpler and easier to deal with, it is the way many scientists have been trained. There are however two problems with deterministic approach, first we can not measure with absolute certainly the current state of any system, and second, even if we treat a system classically (ignoring quantum effects), a deterministic interpretation is only valid if we take into account *all* the degrees of freedom. Say, we have one mole of an idealized classical gas, and we which to predict some future state, to do so deterministically, we need to take into account $2 \times 3 \times 6 \times 10^{23}$ degrees of freedom. This is obviously completely impractical. Deterministically, we are only dealing with a subspace, or projection, so we have to treat it stochastically. Furthermore, consider Ising model, it is well know that if solved deterministically, results in un-physical solutions which are fundamentally wrong.

This chapter will focus developing the theoretical foundations of the Gillespie stochastic simulation algorithm, developing an integrator for the LibRoadRunner library that implements this algorithm, analyzing a simple bistable chemical system deterministically, and stochastically, and will compare and contrast the results of both approaches.

4.2 A Bistable System

Consider the following tri-molecular chemical reaction occurring in a finite volume, with low chemical concentrations:

$$\begin{aligned} A + 2X &\underset{k_2}{\overset{k_1}{\rightleftharpoons}} 3X \\ A &\underset{k_4}{\overset{k_3}{\rightleftharpoons}} 3X. \end{aligned} \tag{4.1}$$

In this reaction, the concentration of substance A is held constant. Physically, this is usually accomplished through the use of a device called a continuously fed stirred tank reactor or CSTR [59]. This device maintains precise concentrations of chemical substances necessary for laboratory experimentation of reactions of the type (4.1). This the simplest reaction that can exhibit bistability [81], and has been extensively studied [25, 31, 32]. Simple bistable

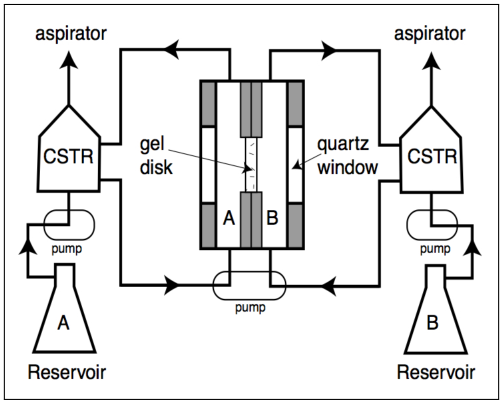


Figure 4.1: A schematic of the gel reactor used by Swinney et al., taken from [59]. The reactions take place in a gel region $2mm$ thick. The gel is confined between two permeable, transparent glass plates. The reservoirs A and B contain "well stirred" mixtures of the reactants, these are continuously pumped through a "continuously fed stirred tank reactor" or CSTR where they are mixed until the concentrations are spatially uniform. The mixtures are allowed to diffuse into the reaction chamber through the permeable glass plates.

reactions of this type are quite rare in nature, in fact, the only physically observed instance of such a simple system was made under exact laboratory circumstances by Swinney et. al [59]. Even though it is rare in nature, it is nonetheless, a physically realizable system, and is the simplest one capable of bistable behavior, thus it provides an ideal system, a *hydrogen atom* if you will for comparing deterministic to stochastic analysis. Deterministically, it will be shown that such as system exhibits a super-critical bifurcation, whereby a single stable fixed point becomes two stable fixed points, and the fixed point to which the system will evolve is determined by the initial conditions. The true dynamics of the system are only evident when the system is analyzed stochastically.

4.3 Deterministic Approach

The bistable system in (4.1) will first be analyzed using the traditional deterministic approach. First, we require a some background which describes the deterministic approach, namely classical reaction rate kinetics.

4.3.1 Deterministic Analysis

Using the law of mass action and the rate constants, we can write a closed form expression for the rate of change of the concentration of X as:

$$\frac{dx}{dt} = -k_2x^3 + k_1Ax^2 - k_4x + k_3A. \tag{4.2}$$

With the appropriate choice of constants,

$$\begin{aligned} B &= k_1A/k_2 \\ R &= k_4/k_2 \\ P &= k_3/k_2. \end{aligned}$$

we can non-dimensionalize (4.2) as:

$$\frac{dx}{dt} = -x^3 + Bx^2 - Rx + BP. \tag{4.3}$$

In order to examine the steady state fixed points of (4.3), we set the left side to zero, and examine the behavior of the right side. We can see that this is a cubic equation which has either one root and two imaginary roots, or three real roots. Thus, we have a system with either one stable fixed point, or two stable and one unstable fixed point, depending on the slope evaluated at the root. The most important parameter in this system is B, which has the effect of shifting the curve up or down, thus determining how many roots exist. Varying the B parameter causes the system to undergo a super-critical pitchfork bifurcation, whereby a single fixed point becomes a single unstable and a pair of stable fixed points.

Such a system when analyzed deterministically only has one or two steady states. There are no limit cycles and there can be no transitioning from one fixed point to another, unless the system is sufficiently perturbed externally to shift the state into the basin of attraction of the other fixed point. The choice of fixed point to which the system will evolve is based entirely on the initial conditions.

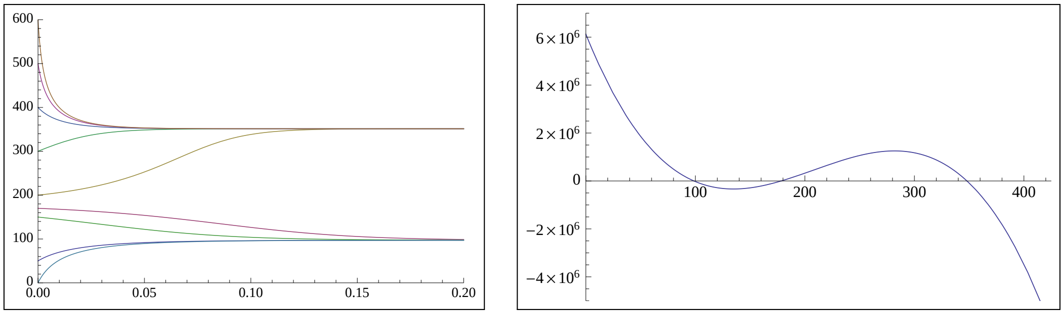


Figure 4.2: numerical simulations of the bistable system using parameters of $B = 625, P = 9800, R = 114000$. The left plot is a superposition of several simulations, each started at a different initial value. The right plot is of the $x' = 0$ nullcline. This system has three real roots at 98.7, 178.3, and 347.9. The first and third roots correspond to the two stable fixed points, and the middle root corresponds to the unstable fixed point.

4.4 Stochastic Approach

In order to analyze system (4.1) stochastically, we need to develop some machinery first, namely the master equation. The following section will follow van Kampen's [79] excellent derivation.

4.4.1 Master Equation

The conditional probability $P_{1|1}(x_2, t_2|x_1, t_1)$, using van Kampen's [79] notation, with the subscript $P_{n|m}$ referring to the probability of n observations given m observations is given by Bayes theorem as the joint probability of observing x_1 at t_1 and x_2 at t_2 is equal to the probability of observing x_1 at t_1 by the probability of x_2 at t_2 given x_1 at time t_1 as

$$P_2(x_1, t_1; x_2, t_2) = P_{1|1}(x_2, t_2|x_1, t_1)P_1(x_1, t_1). \tag{4.4}$$

The conditional probability distribution must of course satisfy the standard probability distribution requirements:

$$\begin{aligned} P_{1|1} &>= 1 \\ \int P_{1|1}(x_2, t_2|y_1, t_1)dx_1 &= 1 \\ P_1(x_2, t_2) &= \int P_{1|1}(x_2, t_2|x_1, t_1)P_1(x_1, t_1)dx_1. \end{aligned}$$

A Markov process is defined as a stochastic process with the property that for any set of n successive times $t_1 < t_2 < \cdots < t_n$, the conditional probability density at time t_n given the observation x_{n-1} at time t_{n-1}, is uniquely determined and is not affected by any observations at previous times:

$$P_{1|n-1}(x_n, t_n|x_1, t_1; \cdots ; x_{n-1}, t_{n-1}) = P_{1|1}(x_n, t_n|x_{n-1}, t_{n-1}). \tag{4.5}$$

Here, $P_{1|1}$ is called the *transition probability.* A Markov process is fully determined by two functions $P_1(x_1|t_1)$ and $P_{1|1}(x_x, t_2|x_1, t_1)$. The entire process for n observations can be constructed from them, e.g. for a three step process, we have:

$$\begin{aligned} P_3(x_1, t_1; x_2, t_2, x_3, t_3) &= P_2(x_1, t_1; x_2, t_2)P_{1|2}(x_3, t_3|y_1, t_1; x_2, t_2) \\ &= P_1(x_1, t_1)P_{1|1}(x_2, t_2|x_1, t_1)P_{1|1}(x_3, t_2|x_2, t_2). \end{aligned} \tag{4.6}$$

For a Markov process, any future state depends only on the present state. It depends neither on the previous history of the process nor on the way in which the present state was reached.

By taking eqn. (4.6) and integrating over x_2 and dividing both sides by P_1, we arrive at the Chapman-Kolmogorov (CK) equation:

$$P_{1|1}(x_2, t_3|x_1, t_1) = \int P_{1|1}(x_3, t_3|x_2, t_2) P_{1|1}(x_2, t_2|x_1, t_1) dx_2. \tag{4.7}$$

The transition probability $P_{1|1}$ does not depend on two different times, only on the difference between the times, so van Kampen introduced the notation

$$P_{1|1}(x_2, t_2|x_1, t_1) \equiv T_\tau(x_2|x_1)\ ,\ \tau \equiv t_2 - t_1, \tag{4.8}$$

where τ is defined as the time interval between times t_2 and t_1. The CK equation then becomes

$$T_{\tau+\tau'}(x_3|x_1) = \int T_{\tau'}(x_3|x_2) T_\tau(x_2|x_1) dx_2. \tag{4.9}$$

The CK equation has a tendency to be inconvenient or difficult to deal with, so it is frequently re-cast in terms of the *master equation.* The master equation is a more convenient form of the CK equation which is obtained by looking at the limit of vanishing time interval difference τ'. If we Taylor expand the transition probability $T_{\tau'}$ about zero for small τ'

$$T_{\tau'}(x_3|x_2) \approx \delta(x_3 - x_2) + \tau' W(x_3|x_2) + \mathcal{O}(\tau'^2). \tag{4.10}$$

As $\tau \to 0$, the delta function indicates that there is a small probability that a state transition occurs, but this probability should tend to zero as $\tau \to 0$. $W(x_3|x_2)$ is the time derivative of the transition probability at $\tau' = 0$, i.e., the transition probability per unit time from $x_2 \to x_3$. In order for this expression to be normalized properly, the integral over x_3 must equal 1, so (4.10) needs a correction term of $(1 - \alpha_0\tau')$ with the delta which indicates that even though there is a small probability that a transition occurs as $\tau' \to 0$, this should not go to zero. Formally, the $(1 - \alpha_0\tau')$ term is the probability that no transition takes place during the time interval τ' so the α_0 coefficient is defined as

$$\alpha_0(x_2) = \int W(x_3|x_2)dx_3. \tag{4.11}$$

Incorporating the correction term, dividing by τ' and taking the limit of $\tau' \to 0$, we arrive at the differential form of the CK equation called the master equation:

$$\begin{aligned} T_{\tau'}(x_3|x_2) &= (1-\alpha_0\tau')\delta(x_3-x_2) + \tau' W(x_3|x_2) \\ \frac{\partial}{\partial\tau}T_\tau(x_3|x_2) &= \int \left[W(x_3|x_2)T_\tau(x_2|x_1) - W(x_2|x_3)T_\tau(x_3|x_1)\right] dx_2. \end{aligned} \tag{4.12}$$

This can be re-written in a more useful form by noting that all transition probabilities are for a given value x_1 at time t_1 as

$$\frac{\partial P(x,t)}{\partial t} = \int \left[W(x|x')P(x',t) - W(x'|x)P(x,t)\right] dx'. \tag{4.13}$$

Finally, if the range of the stochastic variable x is discrete set of states such as an integer number of molecules, we can replace the integral with a summation:

$$\frac{dp_n(t)}{dt} = \sum_{n'} \left[W_{nn'}p'_n(t) - W_{n'n}p_n(t)\right]. \tag{4.14}$$

This form of the master equation makes physical interpretation much clearer, in that we can see that it is a gain-loss equation for the probability of each state. The first term is the gain due to transitions from states n', and the second term is the loss due to transitions into states n'.

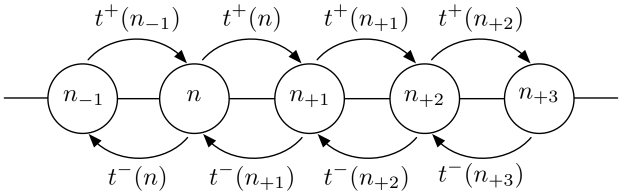


Figure 4.3: Four s steps of a Markov chain, $n_{-1}, n, n_{+1}, n_{+2}, \cdots$ are the states of the chain, and t^+, t^- are the transition probabilities between each state.

Many stochastic processes are a special type called *birth-death* processes, or *one-step* processes. One-step processes describe systems with integer numbers of particles or items, such as individual populations, active neurons, etc. These processes are continuous time Markov processes whose range consists of the integers, $\{x : x \in \mathbb{Z}\}$, and whose transition matrix W only permits jumps between adjacent states,

$$W_{nn'} = t^{+}(x')\delta_{x,x'+1} + t^{+}(x')\delta_{x,x'-1} \tag{4.15}$$

where $t^{+}(x), t^{-}(x)$ are the transition probabilities per unit time of the observation x transition to $x \to x+1,\ x \to x-1$ respectively. The general master equation (4.14) then becomes

$$\partial_t P(x,t|x',t') = t^{+}(x-1)P(x-1,t|x',t') + t^{-}(x+1)P(x+1,t|x',t') - (t^{+}(x) - t^{-}(x))P(x,t|x',t'), \tag{4.16}$$

or the rate of change of the probability of x, t given x', t' is equal to the weighted probabilities of removing a particle plus the probability of adding a particle minus remaining in the same state.

This is called a differential algebraic delay equation, and in general has no analytic closed form solution. It is possible to solve such systems numerically [5], however as of this writing, no known computational mathematics system such as Mathematica, Maple or MATLAB implement such solvers. Under certain circumstances such as a single variable birth-death processes like the bistable system investigated here (4.1) it is possible to write a closed form solution for the stationary distribution of the master equation (4.14).

In order to solve for the stationary probability distribution $P_s(x)$, we re-write the master equation with the *probability current* $J(x)$,

$$J(x) = t^{-}(x)P_x(s) - t^{+}(x-1)P_s(x-1). \tag{4.17}$$

The probability current is anti-symmetric and represents the net flow of probability from $P_s(x)$ to $P_s(x-1)$. As any fool can plainly see, the net flow of probability in a stationary distribution from any given state x to any other state $x + \epsilon$ is zero:

$$J(x+1) - J(x) = 0. \tag{4.18}$$

When we consider that a system can not have a negative number of particles, the negative transition probability (probability of reducing the number of particles) is zero when the particle count is zero:

$$\begin{aligned} t^-(0) &= 0 \\ P(x,t|x',t') &= 0 \;\; \forall\{x < 0 \text{ or } x' < 0\}. \end{aligned} \tag{4.19}$$

Substituting (4.19) into (4.18), and summing (4.18), we have

$$\begin{aligned} J(0) &= t^-(0)P_s(0) - t^+(-1)P_s(-1) = 0, \\ 0 &= \sum_{z=0}^{x-1}(J(z+1) - J(z)) = J(x) - J(0), \end{aligned} \tag{4.20}$$

which implies that the probability current at the x boundary is zero $J(x) = 0$. Thus, we can write a closed form expression for the stationary probability distribution:

$$P_s(x) = \frac{t^+(x-1)}{t^-(x)} P_s(x-1), \tag{4.21}$$

which has the closed form solution of

$$P_s(x) = P_s(0) \prod_{z=1}^{x} \frac{t^+(z-1)}{t^-(z)}. \tag{4.22}$$

4.4.2 Stochastic Analysis

Now that we have all the preliminaries out of the way, we can proceed with expressing the bistable system in master equation form, and analyzing the results.

As the concentration of A is fixed by definition, we are only concerned with the quantity of X particles, we will call this x. There are four reactions involved in this system. The first two create x particles with the forward rate constants of k_1 and k_3, and the second two consume x particles with rate constants of k_2 and k_4. Thus the two transition probabilities are

$$\begin{aligned} t^+(x) &= k_1 Ax(x-1) + k_3 A \\ t^-(x) &= k_2 x(x-1)(x-2) + k_4 x. \end{aligned} \tag{4.23}$$

This pair can be non-dimensionalized using the same set of constants from the deterministic analysis, (4.3). The corresponding deterministic equation is:

$$\begin{aligned} \frac{dx}{dt} &= t^+(x) - t^-(x) \\ &\approx -k_2 x^3 + k_1 A x^2 - k_4 x + k_3 A \qquad (4.24) \\ &= -x^3 + Bx^2 - Rx + BP. \qquad (4.25) \end{aligned}$$

The approximation here was made for $x \gg 1$, setting $x(x-1)(x-2) \approx x^3$. Compare with the deterministic equation derived using the law of mass action (4.3). For large values of x, the master equation, when analyzed deterministically exactly recovers the deterministic equation derived using the law of mass action.

The stationary probability distribution from (4.22) is then

$$P_s(x) = P_s(0) \prod_{z=1}^{x} \left(\frac{B\left((z-1)(z-2) + P\right)}{z(z-1)(z-2) + Rz} \right). \tag{4.26}$$

This is a closed form expression which can be evaluated exactly. The maxima of the stationary probability distribution will occur when the gain and loss transition probabilities are exactly equal, $t^-(x) = t^+(x)$. For the bistable system analyzed here, these occur when

$$Bx(x-1) + P) - (x(x-1)(x-2) + Rx) = 0. \tag{4.27}$$

Note, that for large x, these correspond exactly with the stable fixed points of the deterministic mass action equation (4.3).

4.4.3 Stochastic Simulation and Gillespie's Algorithm

The master equation formulation is analytically solvable under curtain circumstances, and direct numerical solution appears to be quite rare. The most commonly used method for simulating processes that can be described with a master equation is through the use of Gillespie's algorithm [33].

Intuitively, Gillespie's algorithm reflects the fundamental idea that at each step in a single-step birth-death process, some reaction occurs, and that reaction drives the system to the next state. Essentially, Gillespie's algorithm makes the assumption that the probability that the j^{th} reaction will occur once in the next infinitesimal time interval $[t, t+h]$ is approximately equal to $a_j(x)h$, the propensity function, a measure of how likely that particular reaction is to occur. A random number r for each reaction is chosen, and if $a_j(x)h > r$ holds, then we assume the reaction occurs during $[t, t+h]$ and the state of the system are updated according to the state shift vector $\mathbf{v}_j$ corresponding to that reaction as follows $x(t+h) = x(t) + \mathbf{v}_j$.

More formally, Gillespie's algorithm can be derived by considering the evolution of a stochastic processes. A memory-less process is one in which past events have no influence on current events. More precisely, a memory-less process is a stochastic process in which events occur continuously and independently of one another. A frequently used example is radioactive decay. If one observes a sample of radioactive material for some time interval t and either observes or does not observe any events has no influence on whether or not an event will occur in the next time interval δt, i.e. given that there were no events at time t, the probability density between t and $t + \delta t$ is constant, say c, so the probability of observing an event in $[t, t + \delta t]$ is $c \cdot \delta t$. In order to determine the probability of observing an event at time $t_0 + t$, we need to probability density $p(t)dt$ of observing an event between t and $t + \delta t$. We can define the probability density of not observing an event up to time $t + \delta t$ as the probability of not observing an event up to time t and the probability of not observing an event between $[t, \delta t]$, or $P(P_0(t) \cap P_0(t + \delta t))$. For small δt, the probability of not observing an event is $1 - c\delta t$, so with the standard process of letting $\delta t \to 0$, and solving , we have:

$$\begin{aligned} P_0(t+\delta t) &= P_0(t)(1-a\delta t) \\ \frac{P_0(t+\delta t)-P_0(t)}{\delta t} &= -cP_0(t) \\ \frac{d}{dt}P_0(t) &= -cP_0(t) \\ P_0(t) &= e^{-ct}. \end{aligned} \tag{4.28}$$

The cumulative probability density of observing an event between time $[0, t]$ is then $1-P_0(t)$. If no events occurred at time $t = 0$, then $P_0(0)$ is 1. $P_0(t)$ is the probability density of not observing an event up to time t, so the probability density of actually observing an event is then $P(t) = 1 - P_0(t)$. This is however a cumulative distribution function, to get the actual probability of observing an event *at* time t, we need to differentiate it, so,

$$p(t) = \frac{d}{dt}P(t) = ce^{-ct}. \tag{4.29}$$

We now need to determine what is the probability density of the first reaction event. Consider a reaction j which has a rate constant (from the law of mass action) of c_j. The propensity or activity is the the number of ways the reaction can occur. If the reaction is of the kind $A + B \rightarrow$ *something*, then the propensity $c_j n_A n_B$ where n_A and n_B are the number of available items of A and B. If the reaction is of the type $2A \rightarrow$ *something* then the propensity is $\frac{1}{2}c_j n_A(n_a - 1)$. If we call the propensity α_j, then the probability density the the first reaction occurs at time t is:

$$p_j(t) = \alpha_j e^{-\alpha_j t}.w \tag{4.30}$$

Note that the propensity function α_j is time dependent. The propensity changes as a result of reactants being produced or consumed.

Now consider instead of a single event, a system in which there could be $i = 1, \cdots, n$ reactions of type $j = 1, \cdots, r$. If the system started at time $t = 0$, we can call the time at which the first reaction of type j occurs as t_j. In order to calculate the probability density that the first reaction in the system if of type i occurs, we first need to calculate the probability density that a reaction i occurs before any other reaction given that it occurs at time t_i, thus:

$$p(t_i < t_1)p(t_i < t_2)\cdots p(t_i < t_n) = \prod_{j\neq i} p(t_i < t_j) = \prod_{j\neq i} p(t_j > t_i). \tag{4.31}$$

On substitution of the exponential distribution (4.30), we have

$$\prod_{j\neq i} p(t_j < t_i) = \prod_{j\neq i} \int_{t_i}^{\inf} \alpha_j e^{-\alpha_j t} dt = \prod_{j\neq i} e^{-\alpha_j t_i} = e^{-\lambda t_i} e^{\alpha_i t_i}, \tag{4.32}$$

where λ is the total activity of the system given by $\lambda = \sum_{j=1}^{n} \alpha_j$. From here, we can compute the probability density $p(i,t)$ that the reaction i is the first reaction and that it occurs at time t:

$$p(i,t) = e^{-\lambda t} e^{\alpha_i t} \alpha_i e^{-\alpha t} = \alpha_i e^{-\lambda t}. \tag{4.33}$$

Thus given any initial condition, we can now calculate the probability of any reaction i occurring. Gillespie's algorithm iterates over all possible reactions, and chooses the most likely reaction to occur.

Implementation of Gillespie's algorithm

Suppose the state of the system (number of particles of each species, and consequently, the propensity functions α_j) is known at time t. We call s the sum of all $\alpha_j(t)$. Then do the following steps: 1: find the time h after t at which the next reaction will take place, by drawing a random number from an exponential probability density function of rate $p(h) = s\exp(-s\ h)$. Now the most likely reaction which will occur at time $t + h$. 2: Draw a random number from a uniform distribution between 0 and 1. If that number falls between 0 and $a1/s$, choose reaction 1, between $a1/a0$ and $(a1 + a2)/s$ choose reaction 2 and so forth. 3: the chosen reaction is then evaluated. This is accomplished by adding the current state of the system with the state vector that corresponds with the chosen reaction. Thus the values of the α_j (which depend on the particle counts) also change for the next iteration. One then goes back to step 1 of the algorithm with a new distribution of molecules at time $t + h$. The process is reiterated for as long as one wishes to follow the evolution of the system.

Algorithm 1 Implementation of Gillespie's algorithm

```
input $n \leftarrow$ # of iterations
input $r \leftarrow$ # of reactions
input $v \leftarrow$ state shift matrix
input $x \leftarrow$ initial conditions
$t \leftarrow 0$
for $i = 1$ to $n$ do
  $i \leftarrow i + 1$
  $s \leftarrow \sum_{i=1}^{r} a_i(x)$
  $p1 \leftarrow$ uniform random number in $(0, 1]$
  $p2 \leftarrow$ uniform random number in $(0, 1]$
  $h \leftarrow -log(p1)/s$
  $t \leftarrow t + h$
  $cts \leftarrow 0$
  for $j \leftarrow 1$ to $r$ do
    if $ctr < p2 < ctr + a_j/s$ then
      $x(t + h) \leftarrow x(t) + v_j$
    end if
    $ctr \leftarrow ctr + a_j/s$
  end for
end for
```

Gillespie's algorithm was originally prototyped in Mathematica and MATLAB. There are large number of conditional operations, therefore, we should expect performance to be rather sluggish when using interpreted languages. In both languages, Gillespie's algorithm took approximately 5 minutes of wall time for every second of simulation time.

To demonstrate the modular LibRoadRunner architecture, an `GillespieIntegrator` object was created which implements the `Integrator` interface. This was accomplished with about a day's worth of programming effort. Currently, the `GillespieIntegrator` does not yet support SBML events, however this is a planned future addition.

With the native C++ version of the Gillespie SSA, performance improved to approximately 2 - 3 seconds of wall time for each second of simulation. This is a dramatic improvement, and underscores the point that even though interpreted languages have made great strides in the past decade, for any series simulation which consists of non-vectorizable code, one must resort to compiled languages such as C++.

4.4.4 Results

A series of simulations were performed varying B, and holding $P = 9800, R = 114000$ constant. The parameter B has the effect of shifting the probability min / max equation (4.27) up or down. Thus it causes the system to transition from single humped probability distribution to a bimodal distribution. The single humped distribution corresponds to a deterministic equation with a single stable fixed point, and the bimodal distribution corresponds to a deterministic equation with a pair of stable fixed points. For each simulation the system was run for two time units of simulation and was sampled every 0.00005 time units; thus each run has $40,000$ data points.

The results are shown in Figures 4.5 - 4.21. The left plot is of the right-hand side of the deterministic equation (4.3). Again, the zeros of the deterministic equation correspond to the maximum probability values. The center plot contains both the analytic solution of the stationary probability distribution (4.26) (Note, this is a probability density function.) and the data points from the simulation performed using Gillespie's algorithm. All data points from the stochastic simulation were collected and inserted into a histogram. Note that the histogram from the simulation almost perfectly matches the predicted distribution from the master equation. When the simulation was run longer than two time units, the simulation histogram matches the analytic solution even more closely. The right-most column is a time series plot of the simulation. Note that when the system has a unimodal peak, the time series fluctuates about that peak; when the system has a bimodal distribution, the system will fluctuate about both peaks and occasionally *tunnel* between each peak.

It is very informative to compare the results of the stochastic to the deterministic simulation as displayed in Figure 4.4. Note that the probability peaks correspond exactly with

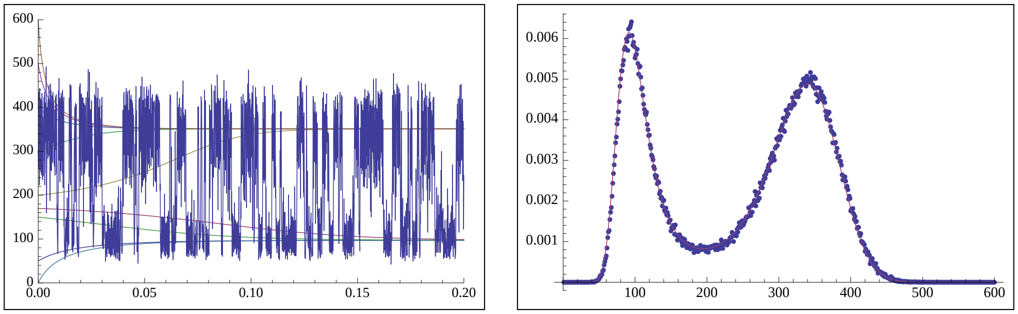


Figure 4.4: numerical simulations of the bistable system using parameters of $B = 625, P = 9800, R = 114000$. The left plot is a superposition of the original deterministic simulation from Figure 4.2, and the time series for the first 0.25 time units of the stochastic simulation. The right plot is the analytic probability density function, and superimposed results of the stochastic simulation.

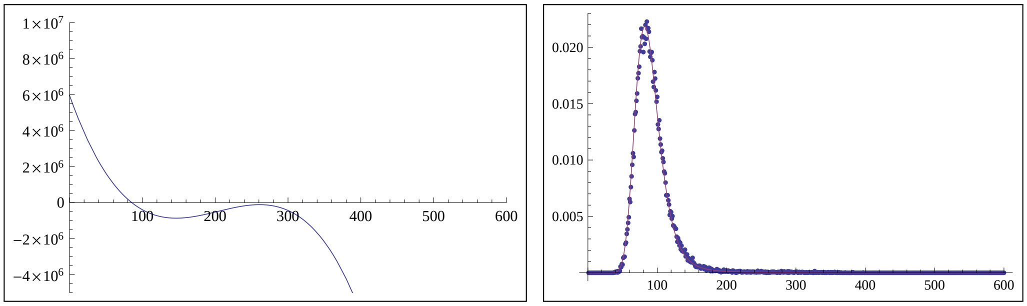


Figure 4.5: nulcline and probability distribution for $B = 608$

the deterministic fixed points.

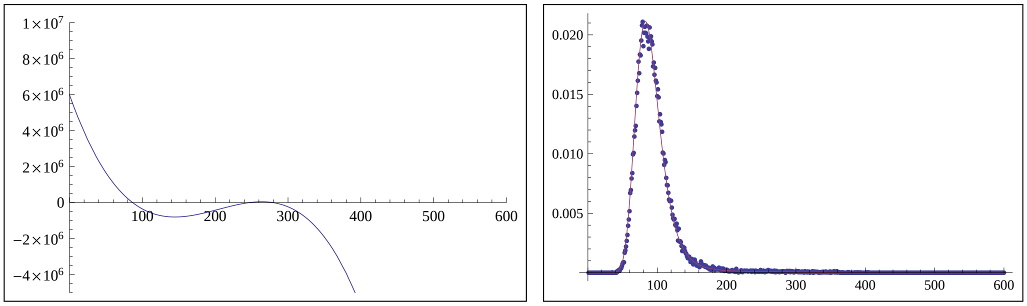


Figure 4.6: nulcline and probability distribution for $B = 610$

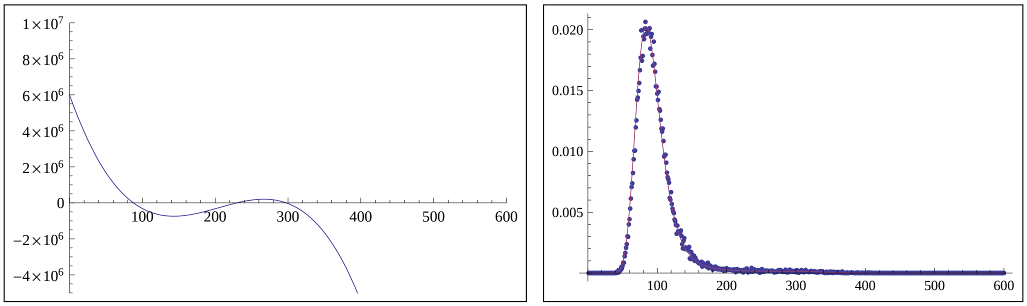


Figure 4.7: nulcline and probability distribution for $B = 612$

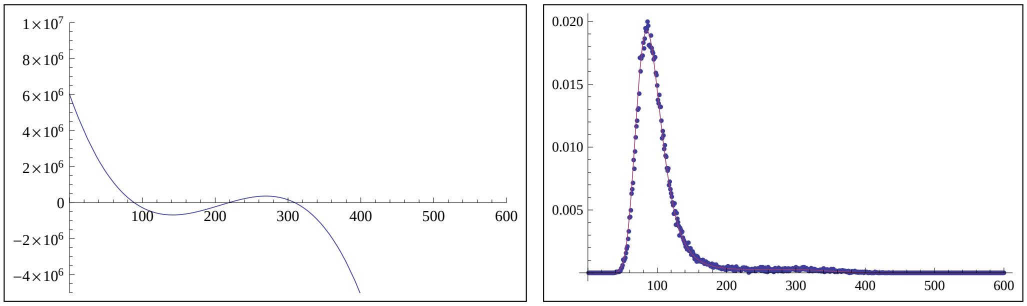


Figure 4.8: nulcline and probability distribution for $B = 614$

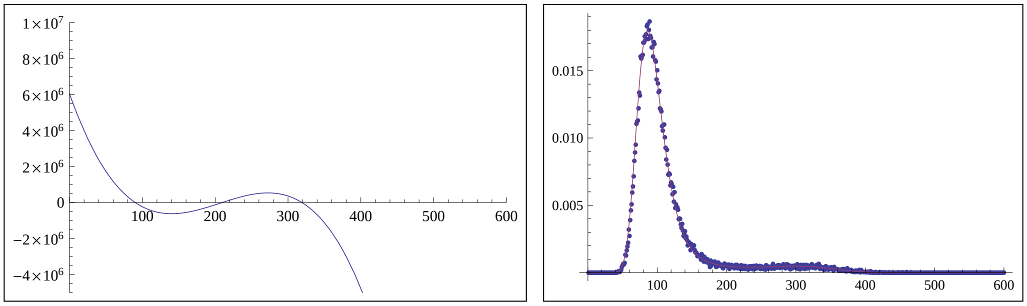


Figure 4.9: nulcline and probability distribution for $B = 616$

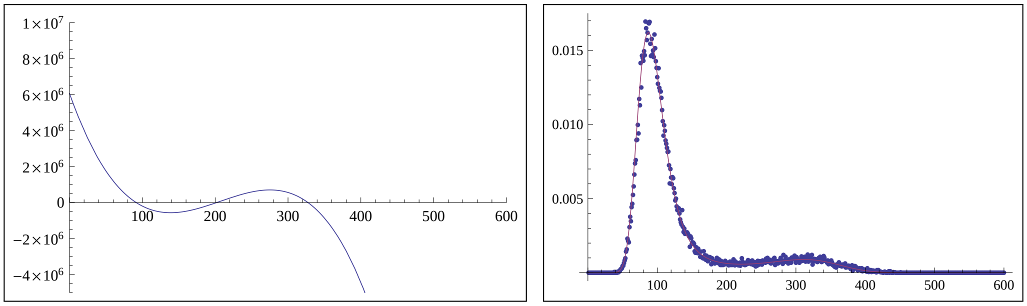


Figure 4.10: nulcline and probability distribution for $B = 618$

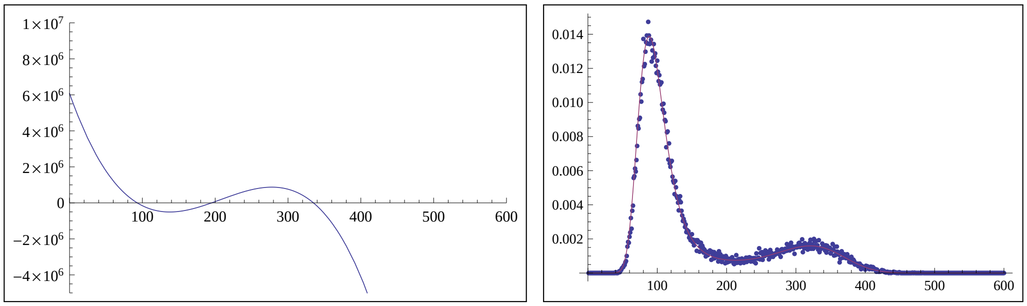


Figure 4.11: nulcline and probability distribution for $B = 620$

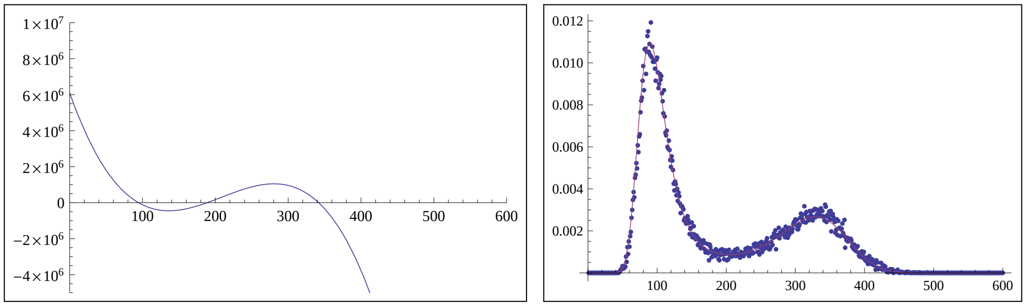


Figure 4.12: nulcline and probability distribution for $B = 622$

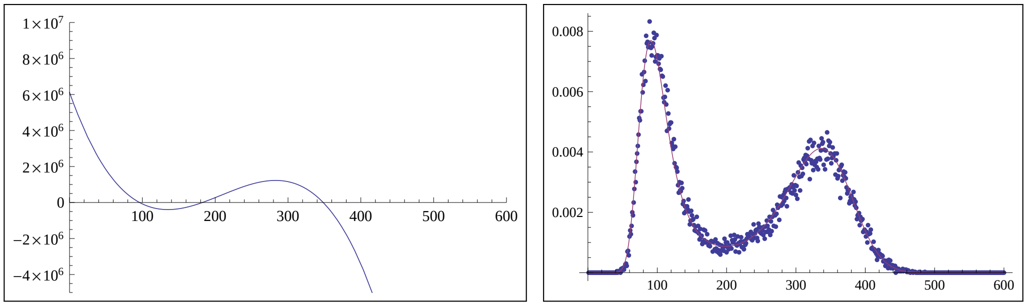


Figure 4.13: nulcline and probability distribution for $B = 624$

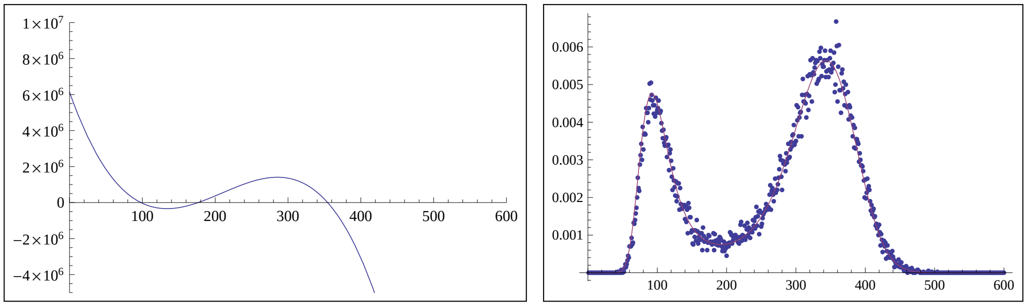


Figure 4.14: nulcline and probability distribution for $B = 626$

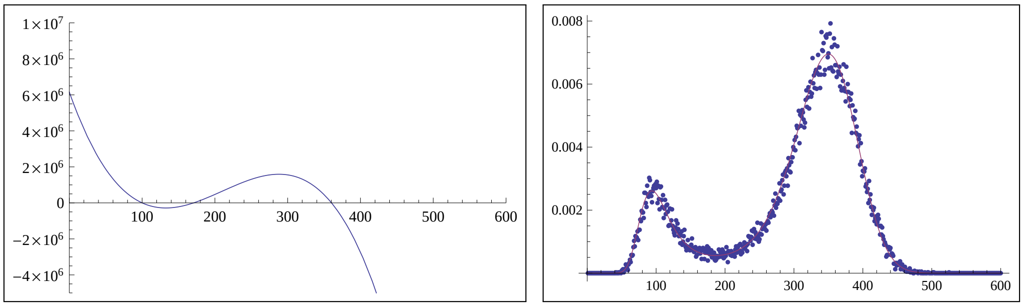


Figure 4.15: nulcline and probability distribution for $B = 628$

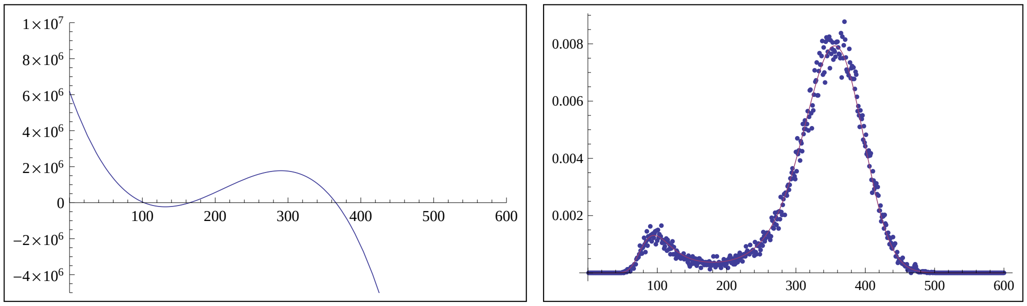


Figure 4.16: nulcline and probability distribution for $B = 630$

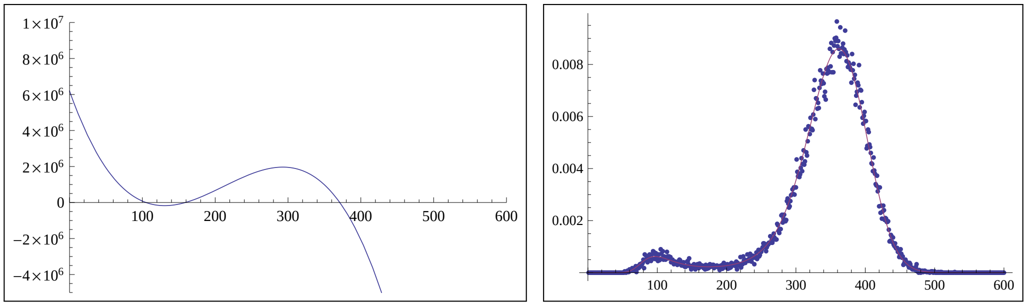


Figure 4.17: nulcline and probability distribution for $B = 632$

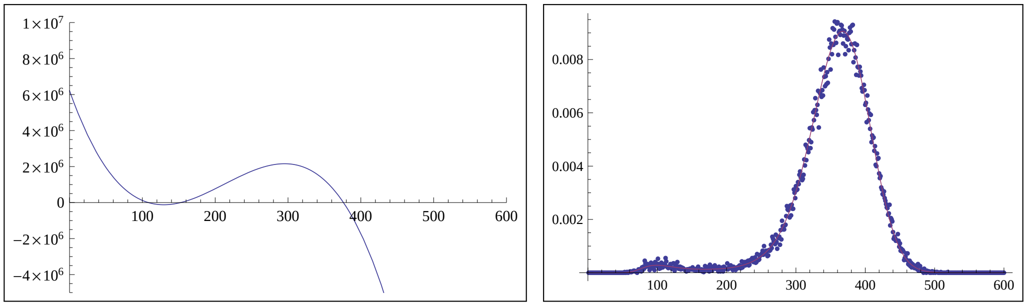


Figure 4.18: nulcline and probability distribution for $B = 634$

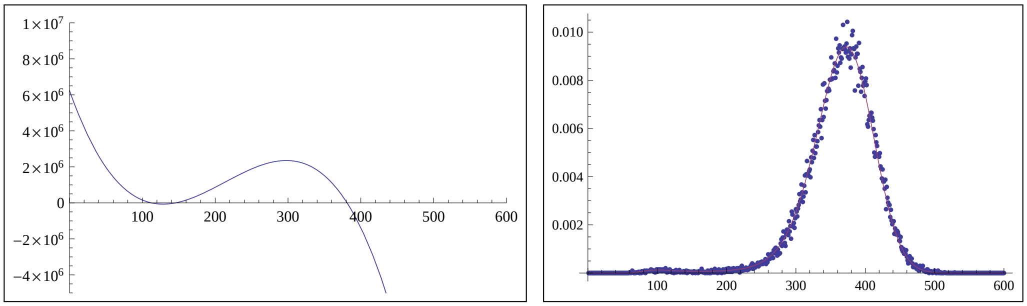


Figure 4.19: nulcline and probability distribution for $B = 636$

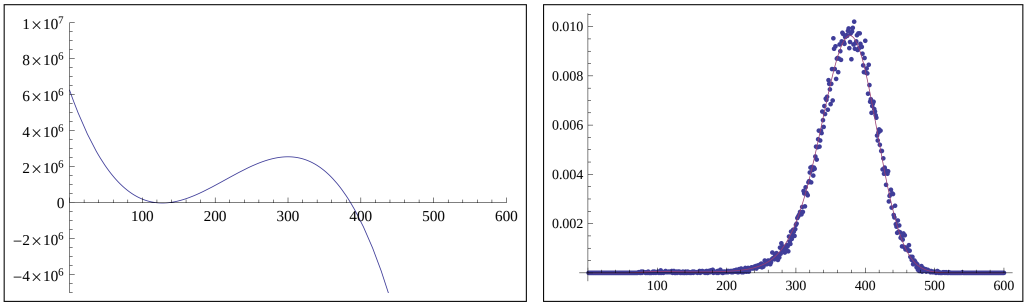


Figure 4.20: nulcline and probability distribution for $B = 638$

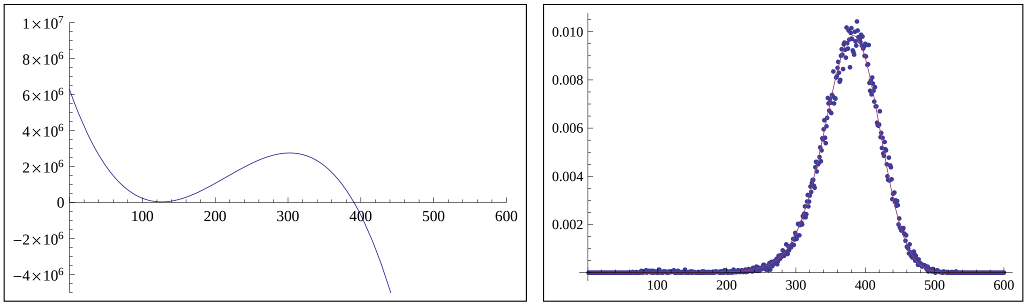


Figure 4.21: nulcline and probability distribution for $B = 640$

4.4.5 Detrended Fluctuation Analysis

Detrended fluctuation analysis (DFA) is a variation of classic root-mean square analysis of a random walk. It is used for determining the long-range correlations (or self affinity) in time series. This method has several advantages over other forms of scaling analysis; chiefly, it is less susceptible to noise effects. Briefly, the time series of length N that is to be analyzed is first integrated, which converts a bounded time series x_t into an unbounded process X_t:

$$X_t = \sum_{i=1}^{t}(x_i - \langle x \rangle). \tag{4.34}$$

Then the integrated time series is divided into time windows of equal length, n. In each window of length n, a least squares line, which represents the trend in that window, is fit to the data. The slope and y intercept of the least squares fit are denoted a_n and b_n respectively. The y coordinate of the straight line segments is denoted by $y_n(t) = a_n t + b_n$. Next the integrated time series X_t is detrended by subtracting the local trend, $y_n(t)$, in each window. The root-mean-square fluctuation of the integrated and detrended time series is calculated by

$$F(n) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(X_i - y_n(i))^2}. \tag{4.35}$$

This evaluation is repeated over the whole signal at a range of different time scales (window sizes) n to characterize the relationship between $F(n)$, the average fluctuation and the window size, n. A log-log plot of n and $F(n)$ is then created. The scaling exponent α is calculated as the slope of a least squares fit of $\log(n)$ vs. $\log(F(n))$. Typically, $F(n)$ will increase with window size. A linear relationship on a log-log plot indicates the presence of power law (fractal) scaling. A straight line on this graph (i.e., low error in the least squares fit) indicates a high statistical self-affinity.

The value of α determines the character of the time series. These are summarized as:

$\alpha < 1/2$: anti-correlated

$\alpha \approx 1/2$: uncorrelated, white noise

$\alpha > 1/2$: correlated

$\alpha \approx 1$: $1/f$-noise, pink noise

$\alpha > 1$: non-stationary phenomena such as random walk or unbounded Brownian noise.

The stochastic simulation is driven by a pair of random uncorrelated numbers, and thus can be considered white noise. This section of analysis seeks to determine whether the type of noise observed in the simulated time series is correlated: i.e., is it $1/f$ or pink noise which has an $\alpha \approx 1$. This type of noise is said to have long range correlations, where the value at a particular step is strongly correlated with the previous steps. This long term correlation is an indication that information at previous times is being propagated forward in time. Pink noise is in contrast to white noise, where the value at any step is completely uncorrelated with any value at previous steps.

In order to examine how well, if at all the system is self correlated, the B parameter was varied from 575 to 675, and all other values were held constant. This range of B clearly transitions the system from a unimodal to bimodal, and back to a unimodal probability distributions. The measured α remained very close to 0.5 for unimodal distributions, and approached 1 for the bimodal distributions. This is an indicator that the system has no self affinity when it is a unimodal state, and has a high degree of self affinity when in a bimodal state. These results were completely unexpected, as to our knowledge, this type of analysis has never been performed before.

The results of the DFA can be seen in Figure 4.22

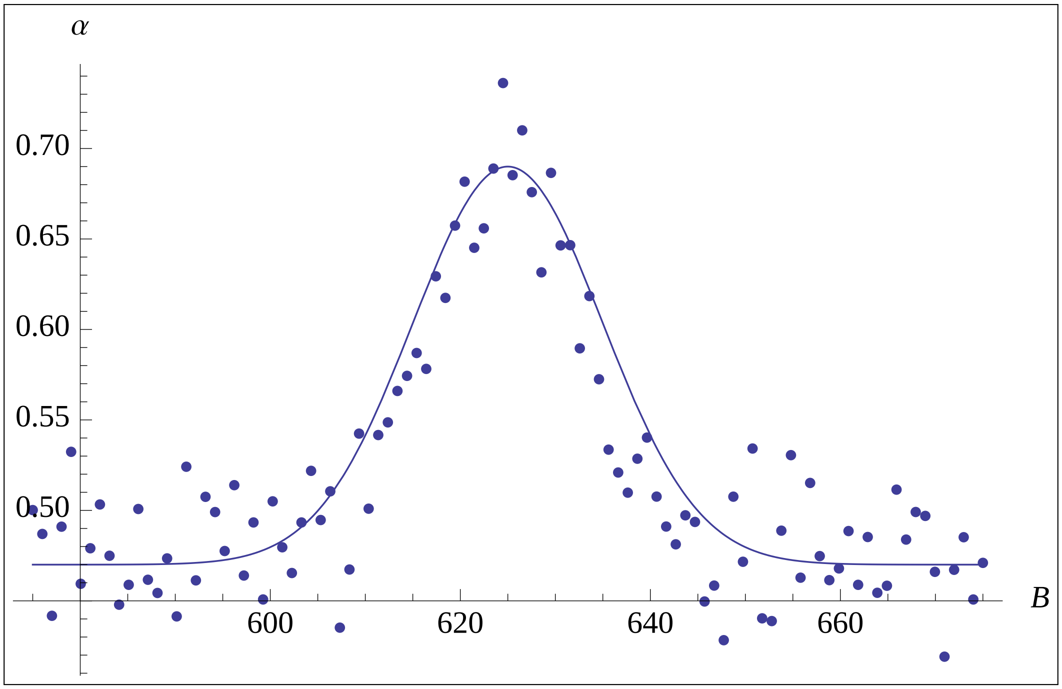


Figure 4.22: A plot of the α exponent from the detrended fluctuation analysis. Here, the B parameter was varied from 575 to 675, while the other two parameters, $P = 9800, R = 114000$. were held constant. A Gaussian was fitted to the α exponents.

4.5 Lattice Lotka Volterra

A series of lattice stochastic simulations were also run. In these type of simulations, each lattice site contains an instance of a stochastic system, that is evolved with Gillespie's algorithm. The equation describing each site is

$$\begin{aligned}\frac{dg}{dt} &= bg(t)\left(1 - \frac{g(t)}{c_1}\right) - d_g g(t) - pg(t)r(t)\\ \frac{dr}{dt} &= e\,p\,g(t)r(t) - d_r r(t),\end{aligned}$$

where g and r are the populations of prey and predator respectively, b is the prey birth rate, c_1 is the carrying capacity, d_g is the natural prey death rate, p is the conversion, or predation rate, and d_r is the predator death rate. Each lattice site is coupled to its nearest neighbors via a diffusion Monte-Carlo algorithm. Preliminary results were presented in class. An initial investigation indicates that there is a clear phase transition as the diffusion constant, or the conversion rate is varied. These preliminary results indicate the need for further analysis.

4.6 Conclusions

This chapter developed the basic governing equation, the master equation for stochastic dynamics, presented an algorithm for simulating stochastic dynamics, compared the simulated results favorably with the analytic results. This chapter has also compared the results of stochastic analytics and simulation to deterministic analytics and simulation, and has shown that the deterministic approach is often only an approximation to stochastic results, and in the case of the bistable system, the deterministic approach is not capable of representing the full dynamics of the system.

A detrended fluctuation analysis was also performed on the bistable stochastic simulation which indicated that the system must have a bimodal probability distribution in order for the system to display $1/f$ or pink noise colored output. To my knowledge, this type of analysis has never been performed on this particular stochastic dynamical system. The prevalence of $1/f$ noise in nature might indicate that the underlying system has a bimodal probability distribution, or at least has higher modality than one.

A framework has been developed to allow large scale simulations of many coupled stochastic dynamical systems on distributed memory supercomputers. At the time of this writing, only a cursory analysis has been performed on these results, however the key point is that a framework now exists for future work.

Chapter 5

Conclusions and Future Work

Despite having been publicly available for a relatively short time as the time of writing, the LibRoadRunner library is already being used at a diverse set of places including:

- University of Washington. The LibRoadRunner library is used here as a key component of the Tellurium project [40], a MATLAB like environment for systems biology.
- Indiana University. The LibRoadRunner library replaced the SOSLib SBML engine in CompuCell3D, which resulted in a 10 fold performance improvement and significantly simplified code.
- University of Southern California. The LibRoadRunner library is enabling heterogeneous modeling and large scale parameter optimization in a multi-scale synaptic and neuronal modeling.
- Charité - Universitätsmedizin Berlin
- Stellenbosch University, South Africa, JJJ Group for Molecular Cell Physiology

To the best of our knowledge, the LibRoadRunner library developed here is the first SBML simulation engine which supports true JIT compilation. This results in high performance simulations. The library is the only known engine which allows users to specify arbitrary callbacks which enables heterogeneous modeling and highly customizable simulations. All code is liberally licensed under the Apache license, and along with extensive documentation, is available at [28]

5.1 Jacobian Automatic Differentiation

Numerically calculating the Jacobian when an implicit (stiff) solver is used is one of the most time intensive steps of the simulation process. Numerical estimates the Jacobian can also introduce round-off errors in the discretization process. As the AST exists for the state vector, it would be possible to analytically calculate the Jacobian directly from the AST, however symbolic differentiation symbolic can lean to inefficient code as the number of terms in the symbolic partial derivative are frequently much larger than the original function.

Automatic differentiation (AD) [37] is a numerical technique computing analytic derivatives of programs (rather than mathematical expressions) which solves all of these problems. AD is based on the idea that every programming language provides a limited number of elementary mathematical functions (such as sin, cos, exp, etc.). Every function computed by a program may be viewed as the composition of these intrinsic functions. The analytic derivatives for these elementary functions are known and can be combined using the chain rule. Effectively, AD is a source transformation which yields another function which may be JIT compiled. AD may be more efficient then symbolic differentiation as function generated using AD will have at most a small constant factor more arithmetic operations than the original function.

5.2 Mutable Conserved Moieties

The scheme presented in § 2.5.3 currently works when a user modifies the value of a conserved species, however it does not allow modification of these values from internal SBML events. The method presented in this section will be moved to the JIT generated code produced by the symbol resolver. This will allow events to modify conserved species.

5.3 Better Steady State Solvers

Many sub-cellular reaction networks operate on a very fast time scale relative to intercellular diffusion. When libRoadRunner is used in cellular and virtual tissue simulators, it make simulations more efficient if the sub-cellular reaction network were just driven to a steady state using the current external chemical concentrations. However, the currently used steady state solver, NLEQ [58] has some issues. Because the entire library was designed using a component based design, § 2.3.2, it is very easy to new capabilities such as new steady state solvers. A pair of steady state solvers, one based on Sundials' KINSOL [42], and another one based on COPASI's [44] steady state solver are planned.

5.4 Different Integrators

Currently, the explicit and implicit integrators from the Sundials suite are used in addition to a simple Runge-Kutta integrator and a Gillespie integrator. An integrator based on LSODA is planned as well as the newly developed ARKode [64] integrator which will be part of the next release of the Sundials suite. An integrator based on the banded solver from CVODE will also be developed. In combination with automatic detection of the banded structure of the Jacobian could speed up performance tremendously and would save a significant amount of space for the Jacobian matrix which is especially interesting in the context of very large systems.

5.5 Script extension to use standard scripting languages

The most interesting planned addition is the introduction of the `<script>` tag to SBML via an extension. Arguably the greatest annoyance of using the SBML language is the necessity of specifying functions, rates and mathematical expressions via MathML. It has been 14 years since the first introduction of MathML and the specification is still largely unsupported. The WebKit and Mozilla based browsers have limited support, and all MathML support has been expunged from the Google Chrome based browsers. Mathematica, one of the few applications exports MathML generates MathML that is incompatible with the subset that is officially supported in SBML.

From a user perspective, MathML is *incredibly* verbose, just to specify a basic expression such as "x + 2 + (y * 5)" required approximately 14 lines of MathML. The subset that is supported in SBML is *extremely* limited: officially, the MathML function in SBML do not even support features that have been rudimentary in programming language since the first inception of FORTRAN 57 such as local variables or iteration. Neither are other basic constructs officially supported such as recursion. This opinion is based partially on the fact that author of this dissertation was one the early adopters of MathML as evidenced by the MathML layout and rendering engine I developed in 2003 [27].

It goes without question that SBML is an excellent language for specifying the structure of a biochemical model, much as HTML excels at specifying the structure of a document. But HTML uses the widely used standardized language JavaScript to specify dynamic content. Following the HTML paradigm, we plan on introducing an `<script>` tag to SBML, exactly as is done in HTML where uses may specify functions in a standardized scripting language such as JavaScript or Python. The remainder of the SBML content would be completely unchanged and could reference a JavaScript or Python function exactly the same way as it currently references a MathML function. This tag will be added using the existing SBML extension mechanism to label as an extension.

JavaScript is available in any extant operating system and existing JavaScript or Python interpreters are very simple to embed into existing applications. We will go a step further and will JIT compile the user specified script functions with the current JIT compilation machinery that was developed in this thesis.

Appendix A

Acronyms

SBML Systems Biology Markup Language

JIT Just In Time Compile

L3V1 Level 3, Version 1

CWSC Continuous Well-Stirred Compartments

LLVM Low-Level Virtual Machine

HTML Hypertext Markup Language

RK Reaction Kinetics

FBA Flux Balance Analysis

ETC Electron Transport Chain

API Application programming interface

LHS left hand side

DOM document object model

Bibliography

[1] J Ackermann, P Baecher, T Franzel, M Goesele, and et al. Massively-Parallel Simulation of Biochemical Systems. *GI Jahrestagung*, 2009.

[2] Jeffrey D. Ullman Alfred V. Aho, Ravi Sethi. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co, 1986.

[3] Sushmita L Allam, Viviane S Ghaderi, Jean-Marie C Bouteiller, Arnaud Legendre, Nicolas Ambert, Renaud Greget, Serge Bischoff, Michel Baudry, and Theodore W Berger. A Computational Model to Investigate Astrocytic Glutamate Uptake Influence on Synaptic Transmission and Neuronal Spiking. *Frontiers in Computational Neuroscience*, 6, 2012.

[4] P W Anderson. More is different. *Science*, 1972.

[5] UM Ascher and LR Petzold. The numerical solution of delay-differential-algebraic equations of retarded and neutral type. *SIAM Journal on Numerical Analysis*, 32(5):1635–1657, 1995.

[6] A.J. Bard and L.R. Faulkner. *Electrochemical Methods: Fundamentals and Applications.* Wiley, 2000.

[7] Chérise D Barker, Torsten Reda, and Judy Hirst. The flavoprotein subcomplex of complex I (NADH: ubiquinone oxidoreductase) from bovine heart mitochondria: insights into the mechanisms of NADH oxidation and NAD+ reduction from protein film voltammetry. *Biochemistry*, 46(11):3454–3464, 2007.

[8] David M Beazley et al. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk workshop*, pages 129–139, 1996.

[9] Frank T Bergmann and Herbert M Sauro. SBW - A modular framework for systems biology. In *WSC '06 Proceedings of the 38th conference on Winter simulation*, pages 1637–1645. Winter Simulation Conference, 2006.

[10] BioModels.net Team. BioModels Database. `http://www.ebi.ac.uk/biomodels-main/`, 2014 (accessed Nov 14, 2014).

[11] B. J. Bornstein, S. M. Keating, A. Jouraku, and Hucka M. Extending the libsbml api.

[12] B. J. Bornstein, S. M. Keating, A. Jouraku, and Hucka M. Summary of extension support classes.

[13] Benjamin J Bornstein, Sarah M Keating, Akiya Jouraku, and Michael Hucka. LibSBML: an API library for SBML. *Bioinformatics*, 24(6):880–881, 2008.

[14] Jean-Marie C Bouteiller, Michel Baudry, Sushmita L Allam, RENAUD J GREGET, Serge Bischoff, and Theodore W Berger. MODELING GLUTAMATERGIC SYNAPSES: INSIGHTS INTO MECHANISMS REGULATING SYNAPTIC EFFICACY. *Journal of Integrative Neuroscience*, 07(02):185–197, June 2008.

[15] J.M. Bouteiller, Z Feng, A. Onopa, E. Huang, E.Y. Hu, E. Somogyi, M. Baudry, S. Bischoff, and T.W. Berger. Maximizing predictability of a bottom-up complex multiscale model through systematic validation and multi-objective multi-level optimization. In *Neuroscience 2014*, Washington D.C., 2014. Society for Neuroscience.

[16] U Brandt, S Kerscher, S Dröse, K Zwicker, and et al. Proton pumping by NADH: ubiquinone oxidoreductase. A redox driven conformational change mechanism? *FEBS Letters*, 2003.

[17] Ulrich Brandt. Energy Converting NADH: Quinone Oxidoreductase (Complex I). *Annual Review of Biochemistry*, 75(1):69–92, June 2006.

[18] JAV Butler. The Equilibrium of Heterogeneous Systems Including Electrolytes. Part I. Fundamental Equations and Phase Rule. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, pages 129–136, 1926.

[19] CellDesigner Team. CellDesigner: A modeling tool of biochemical networks. `http://www.celldesigner.org/`, 2014 (accessed Nov 14, 2014).

[20] R.G. Compton and C.E. Banks. *Understanding Voltammetry*. Imperial College Press, 2011.

[21] COPASI Team. COPASI Python API Documentation. `http://www.copasi.org/static/packages/copasi34_python_api.pdf`, 2014 (accessed Nov 17, 2014).

[22] Jun Doi. Peta-scale lattice quantum chromodynamics on a Blue Gene/Q supercomputer. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference*, pages 1–10. IEEE, 2012.

[23] Donnelly, Charles and Stallman, Richard and others. Bison. The YACC-compatible Parser Generator. `http://www.gnu.org/software/bison/`, 2014 (accessed March 5, 2014).

[24] M Dowell and P Jarratt. A modified regula falsi method for computing the root of an equation. *BIT Numerical Mathematics*, 11(2):168–174, 1971.

[25] W Ebeling and L Schimansky-Geier. Stochastic dynamics of a bistable reaction system. *Physica A: Statistical and Theoretical Physics*, 98(3):587–600, 1979.

[26] Rouslan G Efremov and Leonid A Sazanov. Structure of the membrane domain of respiratory complex I. *Nature*, 476(7361):414–420, August 2011.

[27] Endre Somogyi. The gNumerator MathML rendering engine. `http://numerator.sourceforge.net/`, 2014 (accessed March 5, 2014).

[28] Endre Somogyi, Herbert Sauro, et. al. The LibRoadrunner SBML Library. `http://www.libroadrunner.org`, 2014 (accessed March 5, 2014).

[29] Th Erdey-Gruz and M Volmer. Zur theorie der wasserstoffüberspannung. *Z. Phys. Chem. A*, 150:203, 1930.

[30] Dennis H Evans, Kathleen M O'Connell, Ralph A Petersen, and Michael J Kelly. Cyclic voltammetry. *Journal of Chemical Education*, 60(4):290, 1983.

[31] C. W. Gardiner. *Handbook of Stochastic Methods: For Physics, Chemistry and the Natural Sciences (Springer Series in Synergetics)*. Springer, November 1996.

[32] CW Gardiner, KJ McNeil, DF Walls, and IS Matheson. Correlations in stochastic theories of chemical reactions. *Journal of Statistical Physics*, 14(4):307–331, 1976.

[33] D Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 1977.

[34] D.K. Gosser. *Cyclic voltammetry: simulation and analysis of reaction mechanisms*. VCH, 1993.

[35] David C Grahame. The electrical double layer and the theory of electrocapillarity. *Chemical reviews*, 41(3):441–501, 1947.

[36] Renaud Greget, Fabien Pernot, Jean-Marie C Bouteiller, Viviane Ghaderi, Sushmita Allam, Anne Florence Keller, Nicolas Ambert, Arnaud Legendre, Merdan Sarmis, Olivier Haeberle, Michel Faupel, Serge Bischoff, Theodore W Berger, and Michel Baudry. Simulation of Postsynaptic Glutamate Receptors Reveals Critical Features of Glutamatergic Transmission. *PLoS ONE*, 6(12):e28380, December 2011.

[37] Andreas Griewank et al. On automatic differentiation. *Mathematical Programming: recent developments and applications*, 6:83–107, 1989.

[38] H von Helmholtz. Ueber einige gesetze der vertheilung elektrischer ströme in körperlichen leitern, mit anwendung auf die thierisch-elektrischen versuche. *Annalen der Physik*, 165:353–377, 1853.

[39] Hendrik A Heering, Judy Hirst, and Fraser A Armstrong. Interpreting the Catalytic Voltammetry of Electroactive Enzymes Adsorbed on Electrodes. *The Journal of Physical . . .* , 1998.

[40] Herbert Sauro and Michal Galdzicki and Andy Somogyi and Totte Karlsson and Lucian Smith and Stanley Gu and Alex Darling and Nasir Elmi and Kaylene Stocking . Model, simulate, and analyze biochemical systems using a single tool. `http://tellurium.analogmachine.org/`, 2014 (accessed March 5, 2014).

[41] Kathie L Hiebert and Lawrence F Shampine. Implicitly defined output points for solutions of odes. Technical report, Sandia Labs., Albuquerque, NM (USA), 1980.

[42] Alan C Hindmarsh, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, Dan E Shumaker, and Carol S Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005.

[43] S Hoehme and D Drasdo. A cell-based simulation software for multi-cellular systems. *Bioinformatics*, 26(20):2641–2642, October 2010.

[44] Stefan Hoops, Sven Sahle, Ralph Gauges, Christine Lee, Jürgen Pahle, Natalia Simus, Mudita Singhal, Liang Xu, Pedro Mendes, and Ursula Kummer. COPASI—a complex pathway simulator. *Bioinformatics*, 22(24):3067–3074, 2006.

[45] M Hucka, A Finney, H M Sauro, H Bolouri, J C Doyle, H Kitano, , the rest of the SBML Forum, A P Arkin, B J Bornstein, D Bray, A Cornish-Bowden, A A Cuellar, S Dronov, E D Gilles, M Ginkel, V Gor, I I Goryanin, W J Hedley, T C Hodgman, J H Hofmeyr, P J Hunter, N S Juty, J L Kasberger, A Kremling, U Kummer, N Le Novere, L M Loew, D Lucio, P Mendes, E Minch, E D Mjolsness, Y Nakayama, M R Nelson, P F Nielsen, T Sakurada, J C Schaff, B E Shapiro, T S Shimizu, H D Spence, J Stelling, K Takahashi, M. Tomita, J Wagner, and J Wang. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, March 2003.

[46] Michael Hucka, Frank T. Bergmann, Stefan Hoops, Sarah M. Keating, Sven Sahle James C. Schaff, Lucian P. Smith, and Darren J. Wilkinson. The Systems Biology Markup Language (SBML): Language Specification for Level 3 Version 1 Core. http://sbml.org/specifications/sbml-level-3/version-1/core, October 2010.

[47] Roland Keller, Alexander Dörr, Akito Tabira, Akira Funahashi, Michael J Ziller, Richard Adams, Nicolas Rodriguez, Nicolas L Novère, Noriko Hiroi, Hannes Planatscher, Andreas Zell, and Andreas Dräger. The systems biology simulation core algorithm. *BMC Systems Biology*, 7(1):55, July 2013.

[48] H.C. King. *The History of the Telescope.* Dover Books on Astronomy Series. Dover Publications, 2011.

[49] Dilip Kondepudi and Ilya Prigogine. *Modern thermodynamics: from heat engines to dissipative structures.* John Wiley, 1998.

[50] H.P. Langtangen. *Python Scripting for Computational Science.* Texts in Computational Science and Engineering. Springer, 2008.

[51] Christophe Léger, Kerensa Heffron, Harsh R Pershad, Elena Maklashina, César Luna-Chavez, Gary Cecchini, Brian AC Ackrell, and Fraser A Armstrong. Enzyme electrokinetics: energetics of succinate oxidation by fumarate reductase and succinate dehydrogenase. *Biochemistry*, 40(37):11234–11245, 2001.

[52] Christophe Léger, Anne K Jones, Simon PJ Albracht, and Fraser A Armstrong. Effect of a dispersion of interfacial electron transfer rates on steady state catalytic electron transport in [NiFe]-hydrogenase and other enzymes. *The Journal of Physical Chemistry B*, 106(50):13058–13063, 2002.

[53] Albert Lehninger, David L. Nelson, and Michael M. Cox. *Lehninger Principles of Biochemistry.* W. H. Freeman, fifth edition edition, June 2008.

[54] Gilbert Newton Lewis and Merle Randall. *Thermodynamics and the free energy of chemical substances.* McGraw-Hill, 1923.

[55] K Luby-Phelps. The physical chemistry of cytoplasm and its influence on cell function: an update. *Molecular Biology of the Cell*, 24(17):2593–2596, August 2013.

[56] R Machné, A Finney, M. Müller, J Lu, S Widder, and C Flamm. The SBML ODE Solver Library: a native API for symbolic and fast numerical analysis of reaction networks. *Bioinformatics*, 22(11):1406–1407, May 2006.

[57] G. Nicolis and I. Prigogine. *Self-organization in nonequilibrium systems: from dissipative structures to order through fluctuations.* A Wiley-Interscience Publication. Wiley, 1977.

[58] Ulrich Nowak and Lutz Weimann. A family of newton codes for systems of highly nonlinear equations. Technical report, Citeseer, 1991.

[59] Q Ouyang and HL Swinney. Transition to chemical turbulence. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 1:411, 1991.

[60] Terence Parr. *The definitive ANTLR reference: building domain-specific languages.* Pragmatic Bookshelf, 2007.

[61] LR Petzold and AC Hindmarsh. Lsoda. *Computing and Mathematics Research Division, I-316 Lawrence Livermore National Laboratory, Livermore, CA*, 94550, 1997.

[62] GG Powathil, MAJ Chaplain, and M Swat. Investigating the development of chemotherapeutic drug resistance in cancer: A multiscale computational study. *IET Systems Biology*, submitted, 2014.

[63] Torsten Reda and Judy Hirst. Interpreting the catalytic voltammetry of an adsorbed enzyme by considering substrate mass transfer, enzyme turnover, and interfacial electron transport. *The Journal of Physical Chemistry B*, 110(3):1394–1404, 2006.

[64] Daniel R. Reynolds, Carol S. Woodward, David J. Gardner, and Alan C. Hindmarsh. ARKode: A library of high order implicit/explicit methods for multi-rate problems. In *SIAM Conference on Parallel Processing for Scientific Computing*, February 2014.

[65] Julie Theriot Rob Phillips, Jane Kondev. *Physical Biology of the Cell.* Garland Science, 1 edition, 2009.

[66] Herbert M Sauro and Brian Ingalls. Conservation analysis in biochemical networks: computational issues for software writers. *Biophysical Chemistry*, 109(1):1–15, April 2004.

[67] SBML Team. SBML Software Guide. `http://sbml.org/SBML_Software_Guide`, November 2014.

[68] Vitaly A Selivanov, Marta Cascante, Mark Friedman, Mark F Schumaker, Massimo Trucco, and Tatyana V Votyakova. Multistationary and oscillatory modes of free radicals generation by the mitochondrial respiratory chain revealed by a bifurcation analysis. *PLoS Computational Biology*, 8(9):e1002700, 2012.

[69] Vitaly A Selivanov, Tatyana V Votyakova, Violetta N Pivtoraiko, Jennifer Zeak, Tatiana Sukhomlin, Massimo Trucco, Josep Roca, and Marta Cascante. Reactive Oxygen Species Production by Forward and Reverse Electron Fluxes in the Mitochondrial Respiratory Chain. *PLoS Computational Biology*, 7(3):e1001115, March 2011.

[70] Boris M Slepchenko and Leslie M Loew. Chapter One-Use of Virtual Cell in Studies of Cellular Dynamics. *International review of cell and molecular biology*, 283:1–56, 2010.

[71] SOSLib Team. MayLeonardRepressilator. `http://www.tbi.univie.ac.at/~raim/odeSolver/models/mayleonnard_repressilator2_without_rules.xml`, 2014 (accessed March 5, 2014).

[72] SOSLib Team. The SBML ODE Solver Library. `http://www.tbi.univie.ac.at/~raim/odeSolver/news/`, 2014 (accessed March 5, 2014).

[73] SOSLib Team. The SBML ODE Solver Library Preliminary Benchmarking Results. `http://www.tbi.univie.ac.at/~raim/odeSolver/doc/benchmarking.html`, 2014 (accessed March 5, 2014).

[74] A Sucheta, R Cammack, J Weiner, and F A Armstrong. Reversible electrochemistry of fumarate reductase immobilized on an electrode surface. Direct voltammetric observations of redox centers and their participation in rapid catalytic electron transport - Biochemistry (ACS Publications). *Biochemistry*, 1993.

[75] Maciej H Swat, Gilberto L Thomas, Julio M Belmonte, Abbas Shirinifard, Dimitrij Hmeljak, and James A Glazier. Multi-Scale Modeling of Tissues Using CompuCell3D. In *Methods in cell biology*, pages 325–366. Elsevier, 2012.

[76] Hiromu Takizawa, Kazushige Nakamura, Akito Tabira, Yoichi Chikahara, Tatsuhiro Matsui, Noriko Hiroi, and Akira Funahashi. LibSBMLSim: a reference implementation of fully functional SBML simulator. *Bioinformatics*, 29(11):1474–1476, 2013.

[77] M. Tuckerman. *Statistical Mechanics: Theory and Molecular Simulation*. Oxford Graduate Texts. OUP Oxford, 2010.

[78] Ravishankar Rao Vallabhajosyula, Vijay Chickarmane, and Herbert M Sauro. Conservation analysis of large biochemical networks. *Bioinformatics*, 22(3):346–353, 2006.

[79] N. G. van Kampen. *Stochastic Processes in Physics and Chemistry*. North Holland, third edition, May 2007.

[80] Jaime González Velasco. Determination of standard rate constants for electrochemical irreversible processes from linear sweep voltammograms. *Electroanalysis*, 9(11):880–882, 1997.

[81] T Wilhelm. The smallest chemical reaction system with bistability. *BMC Systems Biology*, 3(1):90, 2009.

[82] Yanbing Zu, Richard J Shannon, and Judy Hirst. Reversible, electrochemical interconversion of NADH and NAD+ by the catalytic (Iλ) subcomplex of mitochondrial NADH: ubiquinone oxidoreductase (complex I). *Journal of the American Chemical Society*, 125(20):6020–6021, 2003.

Endre T. Somogyi

andy.somogyi@gmail.com

EDUCATION

Doctor of Philosophy, Physics
Indiana University, Bloomington IN Dec 2014
THESIS - Simulation of Electrochemical and Stochastic Systems Using Just In Time Compiled Declarative Languages

Bachelor of Science, Physics
Metropolitan State College of Denver, Denver, CO Dec 2007
THESIS - A Reactive Lattice Boltzmann Model of Biofilm Self Organization
GPA - 3.9

Bachelor of Science, Computer Science Engineering
University of Colorado, Denver, CO May 2004

ACADEMIC EMPLOYMENT

Associate Instructor
Indiana University Department of Physics / Department of Chemistry Jan 2008 - Dec 2014
Responsible for instructional design and delivery and coordination of recitation; laboratory instruction for both Physics and Chemistry.

Research Associate
Indiana University Center for Cell Virus and Theory Jan 2010 - Dec 2013
Developing the mathematical and computational methods needed to understand the physics and chemistry of supra-million atom molecular assemblies such as virus like particles.

Adjunct Instructor
Metropolitan State College of Denver Department of Physics Jan 2008 May 2008
Responsible for teaching General Physics I courses.

Research Assistant
National Center for Voice and Speech 2008
Develop a computational model of air-flow / soft tissue deformation of the human larynx using the finite element method.

Physics Tutor
Metropolitan State College of Denver Aug 2007- Dec 2007
Tutor for all undergraduate Physics and Mechanical Engineering students

Research Assistant

University of Colorado Emerging Technologies Prototyping Lab 2006 - 2008

Configured an MPI based parallel computing cluster using Linux compute nodes and Apple x-grid controller Developed a distributed Lattice-Boltzmann fluid dynamics simulation

Research Assistant

University of Colorado Computer Graphics Lab Jan 2002 - Aug 2002

Developed interactive deformable surface simulation to research different constraint systems Investigated different techniques to model self-collision Researched polygonization of implicit surfaces

PUBLICATIONS

1. E. T. Somogyi, T. Karlsson, M. Swat, M. Galdzicki, and H. M. Sauro, libRoadRunner: A High Performance SBML Compliant Simulator, biorxiv.org, doi: 10.1101/001230
2. E. Somogyi, A. A. Mansour, and P. J. Ortoleva, DMS: A Package for Multiscale Molecular Dynamics, arxiv-web3.library.cornell.edu, Sep. 2013.
3. H. Joshi, S. Cheluvaraja, E. Somogyi, D. R. Brown, and P. Ortoleva, A molecular dynamics study of loop fluctuation in human papillomavirus type 16 virus-like particles: A possible indicator of immunogenicity, Vaccine, vol. 29, no. 51, pp. 94239430, Nov. 2011.

PUBLICATIONS IN PREPARATION

1. Endre T. Somogyi, Maciej. Swat, Herbert M. Sauro, and James A. Glazier, Dynamic Compilation of Declarative Systems Biology Languages
2. Endre T. Somogyi and P. J. Ortoleva, Electro-Metabolomics Modeling of Mitochondrial Oxidative Phosphorylation

PRESENTATIONS / LECTURES

- A numerical investigation of co-flowing liquid streams using the Lattice-Boltzmann Method, American Physical Society Division of Fluid Dynamics Meeting, Salt Lake City, Utah, November 19, 2007.
- Validation of the Lattice-Boltzmann Method for the simulation of biofilm growth, Poster Session, American Physical Society March Meeting, Denver Colorado, March 5, 2007.
- Numerical Solution of Partial Differential Equations, Guest Lecture, Indiana University, P548 Mathematical Biology, Spring 2008.
- Stochastic Methods in Mathematical Biology, Guest Lecture, Indiana University, P548 Mathematical Biology, Spring 2008.
- Self Organized Criticality in Financial Markets, Guest Lecture, Indiana University, H205, The Self-Organizing Planet, Fall 2012.

INDEPENDENT RESEARCH

gNumerator 2001 - 2005

gNumerator, http://numerator.sourceforge.net, is an open source computer mathematics program to view, edit, evaluate and plot equations in MathML format. It consists of a set of loosely coupled re-usable components:

- MathML DOM (Document Object Model)
- Equation Viewer / Editor
- Numerical Evaluator
- 3-Dimensonal Rendering Component

ADDITIONAL EXPERIENCE

Electronics Experienced in circuit analysis, simulation and fabrication. Experience with field programmable gate arrays micro-controllers.

Computerized Instrumentation Developed several computer programs to control and collect data from various laboratory experiments.

Machining and Fabrication Experienced in design and fabrication of high tolerance laboratory equipment using milling machine, metal lathe, and other machine shop equipment.

Computer Programming and Data Languages Mathematica, Python, Objective - C / C / C++, LabVIEW, Fortran, SmallTalk, Java, SQL, Lisp, Scheme, Pascal, MathML, Assembly (x86, MIPS, Motorola 68k)

PRIVATE SECTOR EMPLOYMENT

Galileo International, Denver, Colorado
Senior Technical Analyst / Statistician 2002-2006
Technical Analyst 2000-2002

- Developed a state space model to predict the behavior of an application server complex under load.
- Developed a statistical model of transaction volume to forecast expected growth profile. This model served as the basis for hardware purchasing decisions.
- Chief architect of an application framework whose functionality is provided by interchangeable plug-ins for an extremely flexible, customizable and maintainable platform.
- Reverse-engineered an interpreter for proprietary scripting language.
- Developed a rules-based system to monitor and evaluate the health of each node in a 75 node cluster.

Sequoia Software, Columbia, Maryland
Software Engineer 1999-2000

- Participated in design and development of a rule based, distributed XML document processing system.
- Developed load balancer to distribute transactions across multiple machines.
- Developed expression evaluator for document routing rules.

Spirit 32 Development, Denver Colorado
Software Engineer 1998 1999

- Developed key portions of a telephone switching system.

National Renewable Energy Laboratory, Golden Colorado
Software Developer 1998

- Re-engineered system level code for Zilog Z-80 based remote data loggers to record meteorological data.
- Designed meteorological data analysis system to read existing data sets, integrate new data and interpolate lacunae.

PROFESSIONAL ORGANIZATIONS

- Student Member, American Physical Society 2006 - present

ADDITIONAL INFORMATION

Citizenship Native-born United States

Foreign Language Skills Fluent in Hungarian