UNIVERSITY OF CALIFORNIA, SAN DIEGO

Computational Methods for Parameter Estimation in Nonlinear Models

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Physics

by

Bryan Andrew Toth

Committee in charge:

Professor Henry D. I. Abarbanel, Chair
Professor Philip Gill
Professor Julius Kuti
Professor Gabriel Silva
Professor Frank Wuerthwein

2011

The dissertation of Bryan Andrew Toth is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2011

DEDICATION

To all those who have helped me along the way.

EPIGRAPH

An Expert:
One who knows more and more
about less and less, until
eventually he knows
everything about
nothing.
—Source Unknown

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Thanks.

VITA

1997	B. S. E. in Engineering Physics *summa cum laude*, University of Michigan, Ann Arbor
2000	M. S. in Mechanical Engineering, Naval Postgraduate School, Monterey
2002	M. S. in Applied Physics, Johns Hopkins University, Baltimore
2006-2010	Graduate Teaching Assistant, University of California, San Diego
2011	Ph. D. in Physics, University of California, San Diego

PUBLICATIONS

Abarbanel, H. D. I., P. Bryant, P. E. Gill, M. Kostuk, J. Rofeh, Z. Singer, B. Toth, and E. Wong, "Dynamical Parameter and State Estimation in Neuron Models", *The Dynamic Brain: An Exploration of Neuronal Variability and Its Functional Significance*, eds. D. Glanzman and Mingzhou Ding, Oxford University Press, 2011.

Toth, B. A., "Python Scripting for Dynamical Parameter Estimation in IPOPT", *SIAG/OPT Views-and-News 21*:1, 1-8, 2010.

ABSTRACT OF THE DISSERTATION

Computational Methods for Parameter Estimation in Nonlinear Models

by

Bryan Andrew Toth

Doctor of Philosophy in Physics

University of California, San Diego, 2011

Professor Henry D. I. Abarbanel, Chair

This dissertation expands on existing work to develop a dynamical state and parameter estimation methodology in non-linear systems. The field of parameter and state estimation, also known as inverse problem theory, is a mature discipline concerned with determining unmeasured states and parameters in experimental systems. This is important since measurement of some of the parameters and states may not be possible, yet knowledge of these unmeasured quantities is necessary for predictions of the future state of the system. This field has importance across a broad range of scientific disciplines, including geosciences, biosciences, nanoscience, and many others.

The work presented here describes a state and parameter estimation method

that relies on the idea of synchronization of nonlinear systems to control the conditional Lyapunov exponents of the model system. This method is generalized to address any dynamic system that can be described by a set of ordinary first-order differential equations. The Python programming language is used to develop scripts that take a simple text-file representation of the model vector field and output correctly formatted files for use with readily available optimization software.

With the use of these Python scripts, examples of the dynamic state and parameter estimation method are shown for a range of neurobiological models, ranging from simple to highly complicated, using simulated data. In this way, the strengths and weaknesses of this methodology are explored, in order to expand the applicability to complex experimental systems.

Chapter 1

Introduction

Nonlinear parameter estimation research is highly fragmented, probably since the discipline is not typically considered its own scientific "field", but instead a collection of tools that are customized to work within separate scientific fields. Part of this is because the general discipline of linear parameter estimation is mature: a few well-defined techniques can more than adequately solve a large number of parameter estimation problems. But this is also because finding general parameter estimation methods that handle the challenges inherent in non-linear dynamical systems is very difficult, especially when dealing with systems that are chaotic. In this thesis, I will describe a general method to tackle the problem of non-linear parameter estimation in dynamical systems that can be described by first-order ordinary differential equations. First, I will define what parameter estimation is and what specific difficulties are involved with non-linear dynamical systems.

1.1 Dynamical Systems

Dynamics is the study of physical systems that change in time, and is the foundation for a wide range of disciplines. The simplest example of dynamics is the kinetic equations of motion that relate position, velocity, and acceleration of an object to an applied force, based on Newton's first Law, F=ma. Dynamical systems generally come in two flavors: a set of differential equations that describe

the time evolution of a group of related variables through continuous time, or an iterative map that gives a mathematical rule to step a system from one discrete time step to the next.

A model describing a dynamical system can be developed in many different ways. Equations can be derived from physical first principles, such as Newton's laws of motion, Maxwell's equations, or a host of others. Measurements of (presumably) relevant dynamic variables and static parameters can be made to "fit" an experimental system to a model. In whichever way a model is constructed, its usefulness is generally predicated on how well it predicts the future state of a physical system.

1.1.1 Linear and Nonlinear Dynamics

A system of differential equations that constitutes a model of an n-dimensional physical system can be written in the form,

$$\begin{aligned} \frac{dy_1(t)}{dt} &= F_1(y_1(t), \ldots, y_n(t), \mathbf{p}) \\ &\vdots \\ \frac{dy_n(t)}{dt} &= F_n(y_1(t), \ldots, y_n(t), \mathbf{p}) \end{aligned}$$

where the functions $\mathbf{F}_i(\mathbf{y}(t), \mathbf{p})$ define a system of governing equations for the model in the state variables, $\mathbf{y}(t)$, with unknown parameters, $\mathbf{p}$. The state $(y_1, y_2, \ldots, y_n)$ is called the phase space, and a time evolving path from some initial state $(y_1(0), \ldots, y_n(0))$ is the phase space trajectory. In general, the $\mathbf{F}_i(\mathbf{y}(t), \mathbf{p})$ can be complicated functions of $\mathbf{y}(t)$, depending on the system being modeled. A system is considered linear if the functional form of $\mathbf{F}_i(\mathbf{y}(t), \mathbf{p})$ depends only on the first power of the $y_i(t)$s. A simple example of a linear system is radioactive decay:

$$\begin{aligned} \frac{dy}{dt} &= F(y(t)) \\ &= -ky(t). \end{aligned}$$

Here y(t) is the number of radioactive atoms in a material sample at time t. This one dimensional system is linear, since the variable y(t) is raised to the power one in the function F(y(t)).

In contrast, in a nonlinear system, the functional form of at least one of the $\mathbf{F}_i$'s includes a product, power, or function of the y_i, as in the equations describing the motion of a pendulum in a gravitational field across an angle θ (Figure 1.1). This motion can be described by:

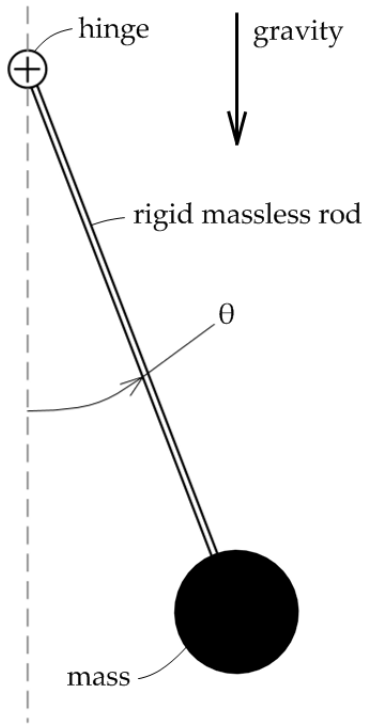

Figure 1.1: Motion of a rigid massless pendulum subject to gravity

$$\frac{d^2\theta}{dt^2} = -sin(\theta)$$

and transformed into our standard form by letting $y_1 = \theta$ and $y_2 = \frac{d\theta}{dt}$:

$$\begin{aligned} \frac{dy_1}{dt} &= y_2 \\ \frac{dy_2}{dt} &= -sin(y_1). \end{aligned}$$

This system is nonlinear, since y_1 appears within the sine function in the second equation. This nonlinearity makes this system very difficult to solve analytically; the solution involves elliptic functions. This textbook example is instead typically

solved usually the small angle approximation, $\sin(y_1) \approx y_1$, which makes the system linear.

In fact, the vast majority of nonlinear dynamical systems cannot be solved analytically, and require linear approximations or numerical solutions. This makes techniques that deal with nonlinear systems very important, since the vast majority of physical systems include some non-linearity. Examples of nonlinear systems include nonlinear electronics, chemical kinetics, lasers, neural networks, economics, ecosystems, and turbulence, to name just a few.

1.1.2 Chaos

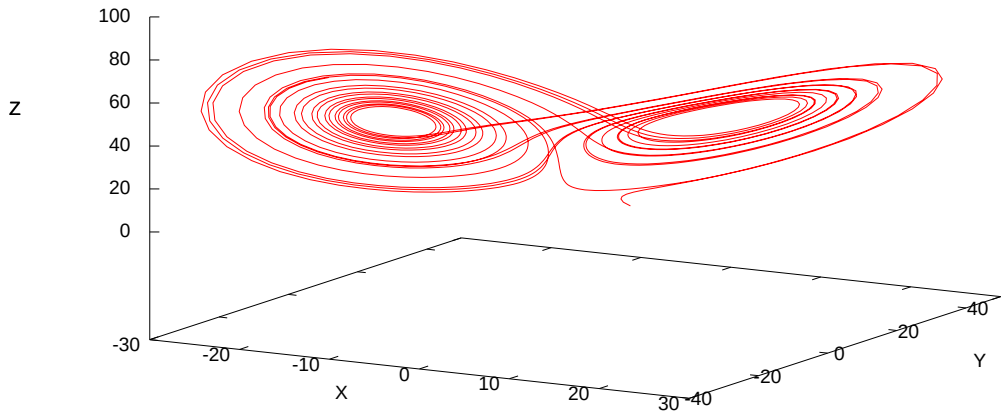

Figure 1.2: Phase space trajectory of the 1963 Lorenz model. Note the characteristic butterfly shape - this is common across a range of parameter values.

The term "chaos" refers to a special class of nonlinear dynamical systems, which are characterized by large changes in the phase space trajectory for small changes in the initial conditions. These changes can be characterized by the Lyapunov exponents associated with the system, which give the average exponential

rate of divergence of initial conditions that are infinitesimally close to each other. If one or more of the Lyapunov exponents are positive, then phase space trajectories from two different initial conditions will diverge, regardless of how close the initial conditions were. The textbook example of a chaotic system is the Lorenz oscillator, first described by Lorenz in 1963[30]:

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x) \\ \frac{dy}{dt} &= rx - y - xz \\ \frac{dz}{dt} &= xy - bz. \end{aligned}$$

Here, σ, r, and b > 0 are parameters. For certain values of the parameters, this system has one positive Lyapunov exponent, and displays chaotic behavior. Figure 1.2 shows a typical phase space trajectory for the Lorenz equations with σ=16, r = 45.92, and b = 4.

This system cannot be solved analytically, and must be solved numerically. Unfortunately, numerical solutions to nonlinear chaotic systems will always diverge from other solutions, with changes to initial conditions, integration algorithm and time step. As a result, long-term prediction is generally impossible for chaotic systems, past some time scale related to the largest Lyapunov exponent of the system. An illustration of this difficulty is presented in Figures 1.3 and 1.4.

In Figure 1.3, the same numerical integration algorithm and integration time step is used to integrate the Lorenz equations with identical parameters and slightly different initial conditions, $(x, y, z) = (10, -5, 12)$ and $(x, y, z) = (10.01, -5, 12)$. The two traces track almost identically until Time=5.5, at which point they diverge.

In Figure 1.4, the same numerical integration algorithm with identical parameters and initial conditions, but integrated with different time steps shows the same divergent behavior, albeit at a slightly later time. Here, divergence occurs at Time=12, beyond which the state variables diverge as before.

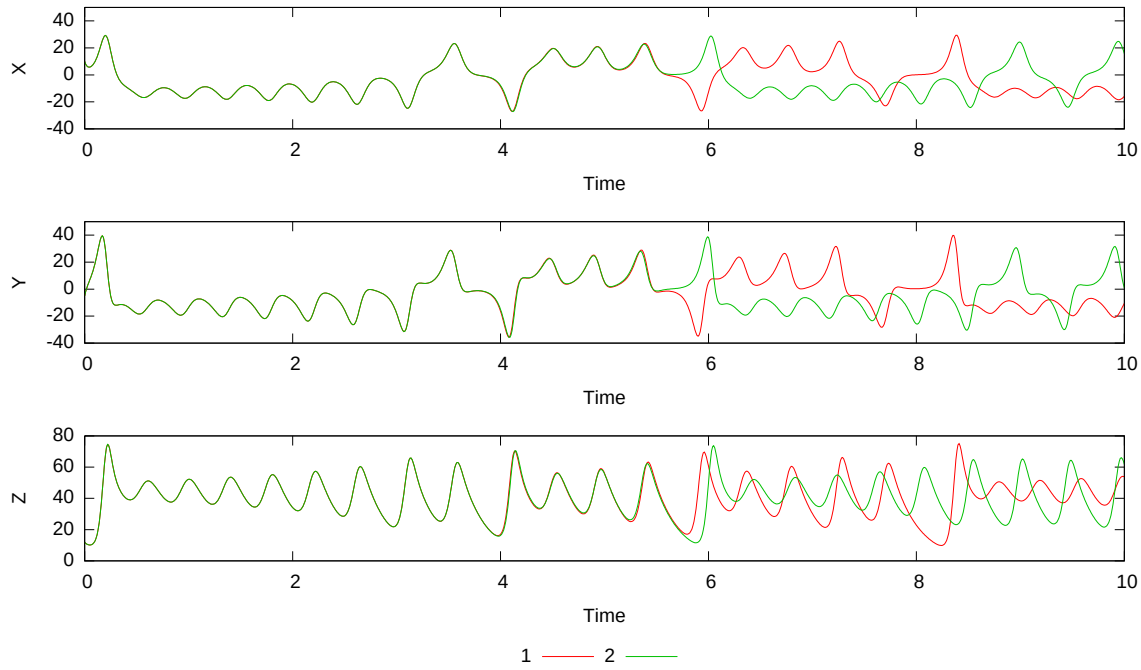

Figure 1.3: Time evolution of the 1963 Lorenz model, integrated with slightly different initial conditions. Trace 1 has initial $x, y, z = 10, -5, 12$ and Trace 2 has initial $x, y, z = 10.01, -5, 12$.

1.1.3 Synchronization

The tendency of chaotic systems to diverge makes them difficult to work with for numerical simulations, particularly in the realm of parameter estimation. To combat this problem, the concept of synchronization in chaotic systems is introduced. Pecora and Carroll [40] argue that chaotic systems can be synchronized when the signs on the Lyapunov exponents of the subsystems are negative. Synchronization is defined as each system converging to the same phase space trajectory and remaining in synchrony. This is accomplished with the addition of an additional term to the dynamics, which effectively couples one signal into the

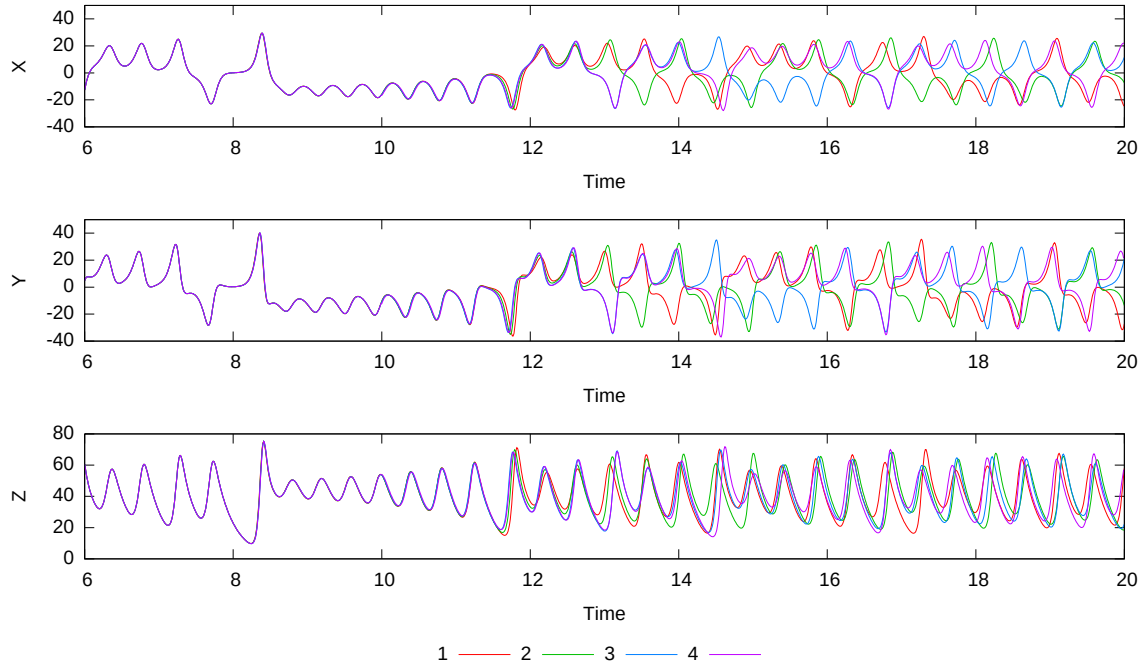

Figure 1.4: Time evolution of the 1963 Lorenz model, integrated at different time steps. From 1 to 4, time steps are 1.0^{-5}, 1.0^{-4}, 1.0^{-3}, 1.0^{-2}. Note that in all cases, all three state variables begin to diverge slightly before time = 12.

other. For the Lorenz oscillator, two identical systems are:

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x) \\ \frac{dy}{dt} &= rx - y - xz \\ \frac{dz}{dt} &= xy - bz. \end{aligned}$$

and

$$\begin{aligned} \frac{di}{dt} &= \sigma(j - i) \\ \frac{dj}{dt} &= ri - j - ik \\ \frac{dk}{dt} &= ij - bk. \end{aligned}$$

If the first equation set is changed to add a coupling term:

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y-x) + K(i-x) \\ \frac{dy}{dt} &= rx - y - xz \\ \frac{dz}{dt} &= xy - bz. \end{aligned}$$

these two systems will synchronize for sufficiently large K.

1.2 Parameter Estimation

In a dynamical system, if all parameters and all state variables at an initial time are known, a prediction can be made as to the future state of the system. This is called the forward problem. Unfortunately, for most experimental systems of interest, only a subset of state variables and parameters can be measured simultaneously, if at all, making prediction of the future state of the system difficult or impossible.

The field of parameter and state estimation, also known as inverse problem theory, is a mature discipline [6, 7, 8, 29, 47, 48, 50, 51] concerned with determining unmeasured states and parameters in an experimental system. This is important since measurement of some of the parameters and states may not be possible, yet knowledge of these unmeasured quantities is necessary for predictions of the future state of the system. This field has importance across a broad range of scientific disciplines, including geosciences, biosciences, nanoscience, and many others.

A subset of the general parameter and state estimation problem is that of data assimilation, where the analysis and forecasts are probability distributions. In this field, two techniques are widely used: Kalman filtering and variational methods (4D-Var).

1.2.1 Kalman Filters

Kalman filters use a Bayesian model to produce a weighted average of predicted and measured values of state variables in a dynamical model, based on measurements of a subset of the state variables. The basic filter has an assumption

of linearity in the system under consideration; another form of the filter, called the extended Kalman filter can be used with nonlinear systems, but since this essentially linearizes the non-linear functions, this does not work well for highly nonlinear and chaotic systems.[15, 16]

1.2.2 Variational Methods

Four dimensional variational data assimilation (4D-Var) [28, 11] is used with increasing frequency in meteorology, combining a short-term numerical forecast with observations. This method adjusts the initial value of the dynamical model instead of changing the state directly. Strong constraint 4D-Var takes available observations during a time window, and seeks a trajectory that best fits those observations. The dynamical parameter estimation technique that we develop here is a special case of strong constraint 4D-Var.

1.2.3 Parameter Estimation in Nonlinear Systems

Both of the above mentioned methods have become widely used in a wide range of disciplines, including meteorology, chemical engineering, and many others. Unfortunately, both methods, in their basic state, rely on linearization of large matrices in the solution algorithm. As a result, these methods perform quite well for linear systems or weakly non-linear systems that can be linearized, but typically perform poorly for systems with strong non-linearities.

Many approaches have been taken to combat this problem, but one direction appears quite promising: the idea of estimating parameters in a nonlinear model by synchronization of experimental observations with a model of the system. This approach has been extensively investigated [33, 39, 40, 41], but specific methods typically still suffer from numerical difficulties during the optimization process. These difficulties stem from using a least-squares metric to measure the difference between measured data and its model equivalent, which has a very complex surface in phase space due to the possibility of chaos [53].

1.3 Dissertation Preview

In this dissertation, I use the idea of synchronization of nonlinear systems as a method of parameter estimation in increasingly complicated neurobiological models. I expand on the methods developed in references [13, 4, 3, 2] and explain some of the benefits and limitations of this parameter estimation methodology with the use of a software package that I developed that streamlines the parameter estimation process. Finally, I explore some of the challenges present in larger dynamical systems, and how to handle them.

Chapter 2

Dynamical State and Parameter Estimation

2.1 Introduction

Dynamical state and parameter estimation (DSPE) [13, 2] is an optimization technique that uses principles from observer theory [37] and synchronization [40] of non-linear systems to determine parameters and unmeasured state variables in an experimental system. DSPE allows a dynamical look into properties of a nonlinear system using only observations of a subset of the dynamical variables, coupled with a mathematical model of the system.

This chapter will discuss the formulation of the dynamical parameter estimation technique, describe an example of the implementation, and give results for this example system.

2.2 DSPE Overview

DSPE is implemented by coupling the output of experimental observations of a system with a model of the system, so that the experimental data and its corresponding model variable synchronize. This is a 'balanced' synchronization between the experimental data and the model, in the sense that the data is trans-

mitted to the model accurately and efficiently, yet the coupling is not so strong that the model becomes overwhelmed by the data. This model-data aggregate is posed as a minimization problem and numerically solved using high performance optimization software, such as SNOPT[22] and IPOPT[54]. Twin experiments, where the "data" has been created using a computational model, have proven the technique for spiking neuron models (e.g. Hodgkin-Huxley), as well as nonlinear circuits, and simple geophysical fluid flow models. DSPE has subsequently been used to examine experimental data from nonlinear oscillator circuits [4, 43].

2.3 Formulation

The state of an experimental system is described by **N** independent dynamical variables, $\mathbf{x}(t) = [x_1(t), x_2(t), \ldots, x_N(t)]$, $\mathbf{q}$ fixed parameters, and perhaps external forcing or inputs to the system. To determine the full state of the system, the $\mathbf{x}(t)$ at some discrete time point are all required, but typically only one or a few components can be measured. If **L** state variables can be observed, then the dynamical variables can be written, $\mathbf{x}(t) = [x_1(t), x_2(t), \ldots, x_L(t), \mathbf{x}_\perp(t)]$, where $\mathbf{x}_\perp(t)$ are the unmeasured state variables. The first-order differential equations describing this experimental system are:

$$\begin{aligned}
\frac{dx_1(t)}{dt} &= G_1(x_1(t), x_2(t), \ldots, x_L(t), \mathbf{x}_\perp(t), \mathbf{q}) \\
\frac{dx_2(t)}{dt} &= G_2(x_1(t), x_2(t), \ldots, x_L(t), \mathbf{x}_\perp(t), \mathbf{q}) \\
&\vdots \\
\frac{dx_L(t)}{dt} &= G_L(x_1(t), x_2(t), \ldots, x_L(t), \mathbf{x}_\perp(t), \mathbf{q}) \\
\frac{d\mathbf{x}_\perp(t)}{dt} &= \mathbf{G}_\perp(x_1(t), x_2(t), \ldots, x_L(t), \mathbf{x}_\perp(t), \mathbf{q})
\end{aligned}$$

2.3.1 Least Squares Minimization

To determine the unmeasured state variables, $\mathbf{x}_\perp(t)$, and fixed parameters, $\mathbf{q}$, a typical solution is a least-squares minimization of the error between the measured data, $x_i(t)$ and a model $y_i(t)$ $(1 < i < L)$; the model state variables $y(t)$ and

parameters $\mathbf{p}$ are defined by:

$$\begin{aligned}
\frac{dy_1(t)}{dt} &= F_1(y_1(t), y_2(t), \ldots, y_L(t), \mathbf{y}_\perp(t), \mathbf{p}) \\
\frac{dy_2(t)}{dt} &= F_2(y_1(t), y_2(t), \ldots, y_L(t), \mathbf{y}_\perp(t), \mathbf{p}) \\
&\vdots \\
\frac{dy_L(t)}{dt} &= F_L(y_1(t), y_2(t), \ldots, x_L(t), \mathbf{y}_\perp(t), \mathbf{p}) \\
\frac{d\mathbf{y}_\perp(t)}{dt} &= \mathbf{F}_\perp(y_1(t), y_2(t), \ldots, y_L(t), \mathbf{y}_\perp(t), \mathbf{p})
\end{aligned}$$

This notation assumes that the model and experimental system are governed by different sets of differential equations, $\mathbf{F}$ and $\mathbf{G}$, with different state variables and parameters to describe each system. In the limit where the model describes the experimental system with infinite accuracy, $\mathbf{F} = \mathbf{G}$.

Least-squares minimization of the error between the measured data, $\mathbf{x}$, and the model, $\mathbf{y}$, minimizes the cost function:

$$C(\mathbf{y}, u, \mathbf{p}) = \frac{1}{2T} \int_0^T \sum_{i=1}^{L} \left\{ \Big(x_i(t) - y_i(t) \Big)^2 \right\} dt$$

by evaluating the minima of $\partial C(\mathbf{y}, u, \mathbf{p})/\partial \mathbf{p}$.

$$\frac{\partial C(\mathbf{y}, u, \mathbf{p})}{\partial \mathbf{p}} = \frac{1}{T} \int_0^T dt \sum_{i=1}^{L} \left\{ \frac{\partial y_i(t)}{\partial \mathbf{p}} (x_i(t) - y_i(t)) \right\} dt$$

2.3.2 Failure of Least Squares Minimization

Least-squares minimization works well for linear models, but does not perform well for non-linear vector fields. This is because the least-squares cost function has multiple local minima, due to the structure of the Jacobian of the vector field.

This is shown in Figure 2.1, which plots the least-squares cost function for a 3-dimensional chaotic oscillator, the Colpitts oscillator, as a function of one (of three) parameter in the model. These data were generated as part of a twin experiment (i.e., computational simulation), so the value of the parameter, η, is known to be 6.2723. As seen in this figure, the cost function does not show a

minimum at the known value of η, and has a significant number of local minima. This is just a cross section of the cost function, since the minimization would be performed across all three parameters and two unmeasured states of the model, but it is illustrative of the difficulty pursuant with least-squares methods on nonlinear models.

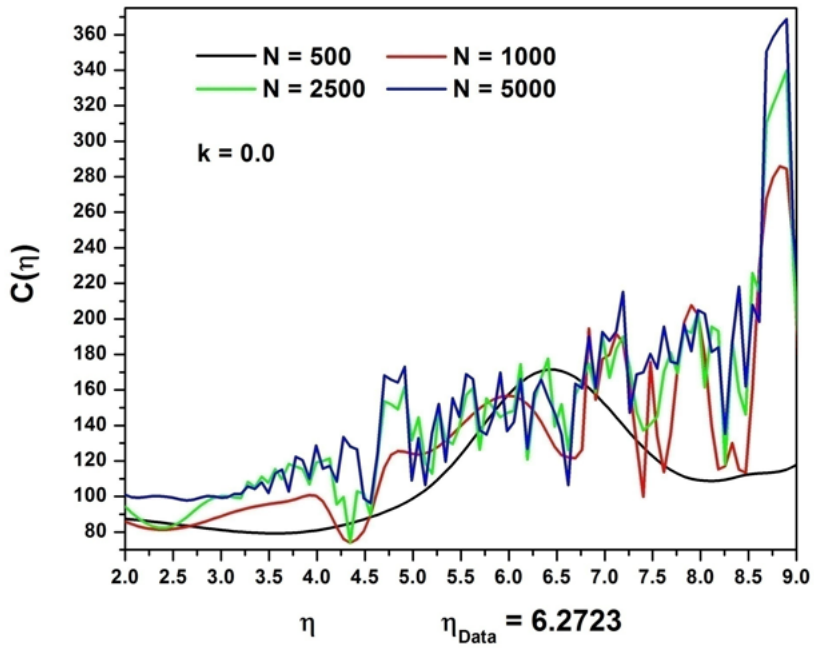

Figure 2.1: The least-squares cost function for the Colpitts oscillator as a function of one of the model parameters. The four curves depict different number of discrete time points, from 500 to 5000. The cost function does not show a minimum at the known value for the model parameter.

This difficulty is expressed mathematically by the structure of the cost function. Specifically, the terms in the minima for $\partial C(\mathbf{y}, u, \mathbf{p})/\partial \mathbf{p}$, $\frac{\partial y_i(t)}{\partial \mathbf{p}}$, are solutions to:

$$\frac{d}{dt}\left(\frac{\partial \mathbf{y}(t)}{\partial \mathbf{p}}\right) = \frac{\partial \mathbf{F}(\mathbf{y}, \mathbf{p})}{\partial \mathbf{y}}\left(\frac{\partial \mathbf{y}(t)}{\partial \mathbf{p}}\right) + \frac{\partial \mathbf{F}(\mathbf{y}, \mathbf{p})}{\partial \mathbf{p}} \tag{2.1}$$

Here, $\mathbf{F}(\mathbf{y}, \mathbf{p})$ is the N-dimensional vector field of the model,

$$\mathbf{F}(\mathbf{y}, \mathbf{p}) = (F_1, F_2, \ldots, F_L, \mathbf{F}_\perp),$$

and $\frac{\partial \mathbf{F}(\mathbf{y},\mathbf{p})}{\partial \mathbf{y}}$ is the NxN Jacobian of the vector field. In a nonlinear system, this Jacobian, iterated along the orbit $\mathbf{y}(t)$, may possess positive global Lyapunov exponents (i.e., the hallmark of chaotic behavior). When positive Lyapunov exponents are present, solutions to equation (2.1) diverge, resulting in the behavior depicted in Figure (2.1).

2.3.3 Addition of Coupling Term

To combat the failure of least squares minimization, dynamical state and parameter estimation couples experimental data to the model system, $\mathbf{y}(t)$ with parameters $\mathbf{p}$, as if for an optimal-tracking problem. This coupling drives the model system to synchronize with the data, and reduces the conditional Lyapunov exponents to non-positive values. This transforms the vector field of the model to:

$$\begin{aligned}
\frac{dy_1(t)}{dt} &= F_1(y_1(t), y_2(t), \ldots, y_L(t), \mathbf{y}_\perp(t), \mathbf{p}) + u_1(t)(x_1(t) - y_1(t)) \\
\frac{dy_2(t)}{dt} &= F_2(y_1(t), y_2(t), \ldots, y_L(t), \mathbf{y}_\perp(t), \mathbf{p}) + u_2(t)(x_2(t) - y_2(t)) \\
&\vdots \\
\frac{dy_L(t)}{dt} &= F_L(y_1(t), y_2(t), \ldots, y_L(t), \mathbf{y}_\perp(t), \mathbf{p}) + u_L(t)(x_L(t) - y_L(t)) \\
\frac{d\mathbf{y}_\perp(t)}{dt} &= \mathbf{F}_\perp(y_1(t), y_2(t), \ldots, y_L(t), \mathbf{y}_\perp(t), \mathbf{p})
\end{aligned}$$

Again, least-squares optimization could be used on the error between the model and the measured data over the course of the time series, but the addition of the coupling terms, $u(t)$, complicates matters. This coupling must be chosen large enough to cause synchronization of the data to the model (and eliminate the positive Lyapunov exponents), but must not overwhelm the underlying dynamics of the system. Addition of the coupling term into the cost function for the optimization ensures that the coupling does not become too large, while appropriate bounds for the range of the coupling ensure that it becomes large enough for synchronization. Therefore, the optimization to be performed is (DSPE):

Minimize:

$$C(\mathbf{y}, u, \mathbf{p}) = \frac{1}{2T}\int_0^T \sum_{i=1}^{L}\left\{\Big(x_i(t) - y_i(t)\Big)^2 + u_i(t)^2\right\}dt$$

Subject to:

$$\begin{aligned}
\frac{dy_1(t)}{dt} &= F_1(y_1(t), y_2(t), \ldots, y_L(t), \mathbf{y}_\perp(t), \mathbf{p}) + u_1(t)(x_1(t) - y_1(t)) \\
\frac{dy_2(t)}{dt} &= F_2(y_1(t), y_2(t), \ldots, y_L(t), \mathbf{y}_\perp(t), \mathbf{p}) + u_2(t)(x_2(t) - y_2(t)) \\
&\vdots \\
\frac{dy_L(t)}{dt} &= F_L(y_1(t), y_2(t), \ldots, y_L(t), \mathbf{y}_\perp(t), \mathbf{p}) + u_L(t)(x_L(t) - y_L(t)) \\
\frac{d\mathbf{y}_\perp(t)}{dt} &= \mathbf{F}_\perp(y_1(t), y_2(t), \ldots, y_L(t), \mathbf{y}_\perp(t), \mathbf{p})
\end{aligned}$$

and also subject to suitable bounds for the state variables and parameters. Suitable bounds for the state variables and parameters are typically chosen based on physical arguments for the given dynamical system.

As shown in Figure 2.2, addition of a coupling term within the model dynamics to synchronize the model to the dynamics results in a cost function cross section that is smooth, with no local minima, and a clear minimum at the correct value of the parameter η. Addition of a coupling term has eliminated the positive conditional Lyapunov exponent in the system, thus regularizing the synchronization manifold of the system.

2.4 Implementation

Since the experimental data is discrete, the cost function integral becomes a summation, and the constraint equations are transformed to a discrete integration map using a numerical integration algorithm, (e.g., Simpson's Rule), with each time step having its own constraint equation for each state variable. The unknown variables for the optimization are each of the state variables and the couplings at each time step and each of the parameters.

The first step is to transform the differential equations into a discrete time map over the interval of interest [0, T]. The choice of numerical integration technique for the differential equations is not unique; Simpson's Rule, with functional

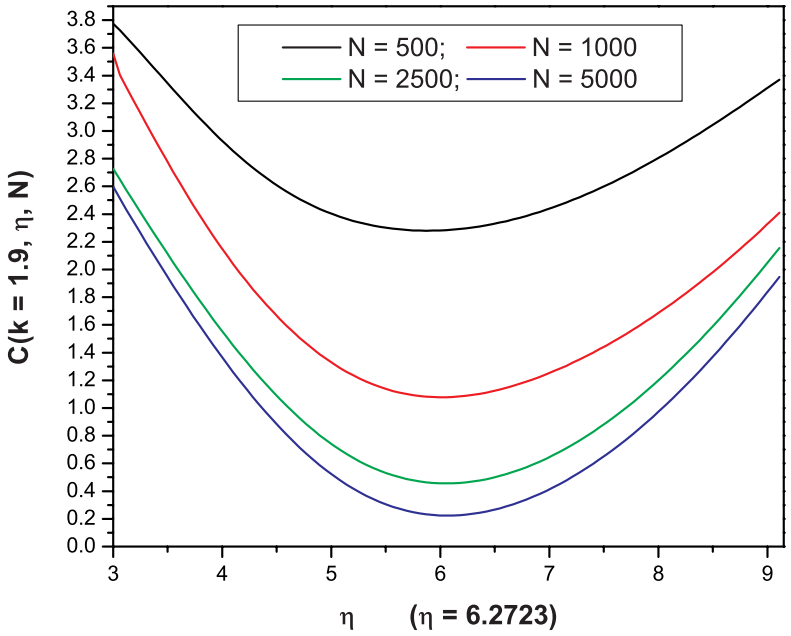

Figure 2.2: The cost function for the Colpitts oscillator as a function of the same parameter as shown previously, but with data coupled to the model equations. Now, the structure of the cost function is smooth, and a clear minimum is evident at the correct value of the parameter.

midpoints estimated by Hermitian cubic interpolation, is chosen here. This choice gives accuracy of order δt^4; methods with similar accuracy will suffice just as well, but lower order methods must be used carefully to ensure that the data sampling rate, $(\frac{1}{\tau})$, is both accurate enough and fast enough to capture the underlying dynamics of the system. These integrated equations become the equality constraints in our optimization:

Simpson's Integration:

$$y_i(n+1) = y_i(n) + \frac{\tau}{6}[F_i(n) + 4F_i(n2) + F_i(n+1)] \tag{2.2}$$

Polynomial Interpolation:

$$y_i(n2) = \frac{1}{2}[y_i(n) + y_i(n+1)] + \frac{\tau}{8}[F_i(n) - F_i(n+1)]. \tag{2.3}$$

where the n2 index refers to the time midpoint between time n and time n+1, and the functions, F_i, include the data coupling term described above. For completeness, a polynomial interpolation is used for midpoints of the control terms as well:

Control Polynomial Interpolation:

$$u_i(n2) = \frac{1}{2}[u_i(n) + u_i(n+1)] + \frac{\tau}{8}[F_i(n) - F_i(n+1)].$$

For a system with 2*T+1 discrete time points, **N** dynamical variables, **L** of which are measured (with control terms), and **p** parameters, this integration scheme gives:

- **N***T Simpson's constraints
- (**N**+**L**)*T Interpolation constraints
- **N***(T+1) + **N***T Dynamical variables
- **L***(T+1) + **L***T Control variables
- **p** Parameter variables.

This defines an optimization space of (**N**+**L**)*(2*T+1)+**p** variables and (2***N**+**L**)*T constraints. In discretized form, the cost function takes the form of a sum, so the optimization problem is:

Minimize:

$$C(\mathbf{y}, u, \mathbf{p}) = \frac{1}{2T} \sum_{t=0}^{T} \sum_{i=0}^{L} \left\{ \Big(x_i(t) - y_i(t) \Big)^2 + u_i(t)^2 \right\} \tag{2.4}$$

subject to the (2***N**+**L**)*T constraints above, with appropriate upper and lower bounds for the (**N**+**L**)*(2*T+1)+**p** unknown variables.

A variety of optimization software and algorithms are available to solve this problem. SNOPT[22] and IPOPT[54] were chosen since they are widely available and are designed for non-linear problems with sparse Jacobian structure. IPOPT can be parallelized with the linear solver Pardiso [44, 45], and can thus solve large problems in a reasonable time. Depending on the problem and the data set, a few thousand data points may be necessary to explore the state space of the model and allow DSPE to produce accurate solutions.

2.5 Example DSPE Problem

2.5.1 Lorenz Equations

As an example of the implementation of the DSPE method, the Lorenz system is chosen. Recall:

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x) \\ \frac{dy}{dt} &= rx - y - xz \\ \frac{dz}{dt} &= xy - bz. \end{aligned}$$

Where σ, r, and b > 0 are parameters. For certain values of the parameters, this system has one positive Lyapunov exponent, and displays chaotic behavior, as shown in Figure 1.2.

Pecora and Carroll, in reference[40], show that synchronization of this system will occur when sufficient coupling is applied to the x or y variable, but not the z variable.

2.5.2 Discretized Equations

Measuring the x-variable and coupling this back to the model using DSPE, mapping (x,y,z) to (y_1, y_2, y_3), this vector field is directly implemented into equations (2.2) and (2.3) to give the optimization constraints. For instance, the y_1 variable constraints are:

Simpson's Integration:

$$
\begin{aligned}
y_1(n+1) = y_1(n) + \\
&\frac{\tau}{6}\Big\{\sigma(y_2(n) - y_1(n)) + u(n)\big(x_1(n) - y_1(n)\big) + \\
&4\Big(\sigma(y_2(n2) - y_1(n2)) + u(n2)\big(x_1(n2) - y_1(n2)\big)\Big) + \\
&\sigma(y_2(n+1) - y_1(n+1) + u(n+1)\big(x_1(n+1) - y_1(n+1)\big)\Big\}
\end{aligned}
$$

and

Polynomial Interpolation:

$$
\begin{aligned}
y_1(n2) = \frac{1}{2}[y_1(n) + y_1(n+1)] + \\
&\frac{\tau}{8}\Big\{\sigma(y_2(n) - y_1(n)) + u(n)\big(x_1(n) - y_1(n)\big) - \\
&\sigma(y_2(n+1) - y_1(n+1)) + u(n+1)\big(x_1(n+1) - y_1(n+1)\big)\Big\}
\end{aligned}
$$

with similar constraints for the other two variables, with no coupling terms.

2.5.3 Cost Function

In discretized form, the cost function takes the form of a sum, so the optimization problem is:

Minimize:

$$C(\mathbf{y}, u, \mathbf{p}) = \frac{1}{2T}\sum_{t=0}^{T}\Big\{\Big(x_1(t) - y_1(t)\Big)^2 + u(t)^2\Big\}$$

This cost function is subject to the constraints above, which occur at every time point. With T time points, there are then 6T constraints. The optimization is also constrained by appropriate upper and lower bounds for the 8T+7 unknown variables.

2.6 Example Results

The purpose of DSPE is to find parameters and states from experimental systems. In order to test any method for this, a twin experiment is first performed; instead of experimental data, twin data is numerically generated for a known set of parameters and initial conditions. Because the "unknown" parameters are actually known in this scenario, the twin experiment gives a clear indication of the viability of the method for a given system. For the Lorenz system, a twin experiment with 5000 data points was run. The x_1-variable "data" was coupled into the experimental system, and the three unknown parameters, as well as the two unmeasured state variables were calculated exactly, as shown in Table 1 and Figures 1 and 2.

An important check is to ensure that the coupling u(t) terms become small as a result of the optimization. Since the synchronization term, $u(t)(x_1(t) - y_1(t))$, is not part of the real dynamical system, this term should be minimized in the dynamics. This is confirmed with the introduction of the 'R-value', which measures the relative contributions of the equation dynamics, $F_1(y_1(t), \mathbf{y}_\perp(t), \mathbf{p})$, and the synchronization term, $u(t)(x_1(t) - y_1(t))$. Formally, the R-value measure is defined as the ratio:

$$R - value \quad = \quad \frac{[F_1(y_1(t), \mathbf{y}_\perp(t), \mathbf{p})]^2}{[F_1(y_1(t), \mathbf{y}_\perp(t), \mathbf{p})]^2 + [u(t)(x_1(t) - y_1(t))]^2}$$

An R-value is calculated for each equation with a synchronization term. An R-value of one at every time point indicates that the optimization found a solution with minimal coupling, while an R-value which varies significantly from one indicates that a suitable optimization fit was not made between the data in the model, which may be an indication that the model incorrectly describes the experimental data. For the Lorenz twin experiment, only one R-value is necessary as only one state variable is coupled; in this instance it was calculated to be 1.00 at all time points as expected.

Table 2.1: Lorenz Model: Data Parameters and Estimated Parameters.

Parameter	Data	Result
σ	10	10.0
r	2.67	2.67
b	28	28.0

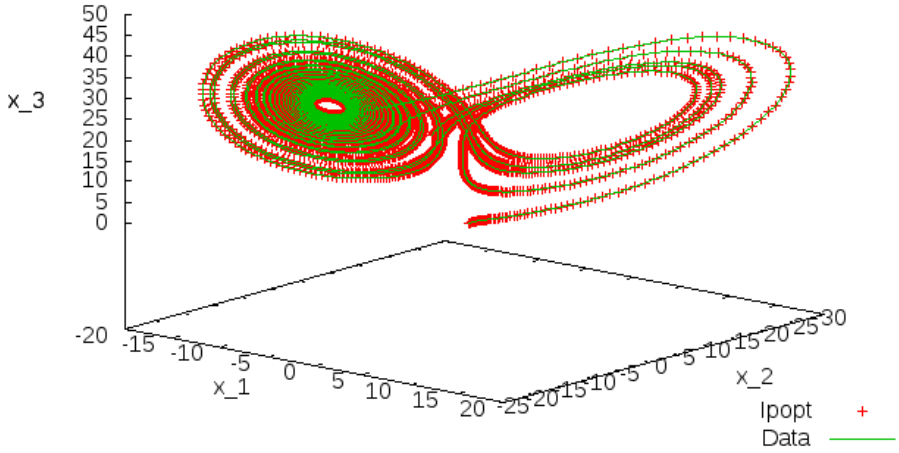

Figure 2.3: 3-D view of dynamical variable results for the Lorenz system dynamical parameter estimation problem.

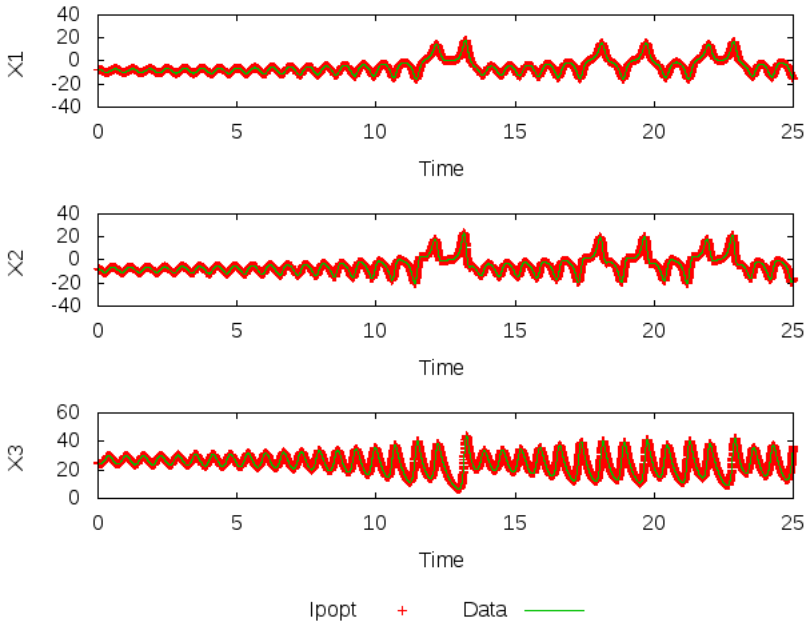

Figure 2.4: Dynamical variable results for the Lorenz system dynamical parameter estimation problem.

Chapter 3

Path Integral Formulation

The Dynamical State and Parameter Estimation technique performs admirably for nonlinear models in twin experiments, but has a significant weakness when applied to real experimental systems. Real experimental systems involve multiple sources of stochastic effects (i.e, noise), which cannot be easily modeled in the deterministic construction of DSPE. This noise can take many forms,

- Systematic: Instrument errors
- Random: Environmental variations
- Model: Incorrectly modeled physics,

but generally must be accounted for in any theoretical model of the system.

The optimization software that implements DSPE in general requires differentiable functions; most noise sources cannot be described by functions at all, much less differentiable functions. In order to include the effects of noise in an experiment and in a model, a different method must be used, starting with stochastic differential equations.

3.1 Stochastic Differential Equations

Stochastic differential equations include terms that model stochastic (i.e., random) processes, thus giving a stochastic (non-deterministic) solution. The sim-

plest stochastic process can be described by a string of random numbers, with no number reliant on the values of any other. A stochastic process does not have a deterministic time evolution, but instead is described by a probability distribution of possible states. Stochastic differential equations are described by a Langevin equation, which in turn can be used to derive a Fokker-Planck equation, which is typically described by[25]:

$$\frac{\partial}{\partial t}P(y,t) = -\frac{\partial}{\partial y}A(y)P + \frac{1}{2}\frac{\partial^2}{\partial y^2}B(y)P$$

where A(y) and B(y) are real differentiable functions with the restriction that $B(y) > 0$, and P(y,t) is the probability distribution function of the state of a given system. As show in reference[55], the Fokker-Planck equation can be replaced by a path integral formulation of the form:

$$P(y(t_m)) = \int \sum_{n=0}^{m-1} d^D y(n) exp[-A_0(\mathbf{Y})] \tag{3.1}$$

where $A_0(\mathbf{Y})$ is called the action of a system, and is a function of the path of the dynamical system, $\mathbf{Y} = \{\mathbf{y}(m), \mathbf{y}(m-1), \ldots \mathbf{y}(0)\}$. Here, the path is identical to a phase space trajectory of the system. The most probable path, corresponding to the deterministic solution to the model equations, maximizes $P(y(t_m))$, thus minimizing the action. Expectation values of quantities related to the path of the vector field can be calculated from this probability distribution; to do this, the action must be derived with respect to model system dynamics and experimental measurements.

3.2 Approximating the Action

3.2.1 Assumptions

In order to derive the action, the following assumptions are made:

- Markovian dynamics. The state $\mathbf{y}(t_{n+1}) = \mathbf{y}(n+1)$ depends only on the state of the system at previous time t_n.
- All noise terms, both in measurement and model, are Gaussian.

- The noise in each measurement is independent of each other measurement in time.

3.2.2 Derivation

As discussed in reference [52], the derivation of the action begins with the definition of conditional probability. The conditional probability for a system to be in state $\mathbf{y}$, conditioned on measurements $\mathbf{X}$, is denoted by $P(\mathbf{y}|\mathbf{X})$, and is formally equal to the probability that a system is simultaneously in the states $\mathbf{y}$ and $\mathbf{X}$ divided by the probability that the system is in the state $\mathbf{X}$.

$$P(\mathbf{y}|\mathbf{X}) = \frac{P(\mathbf{y}, \mathbf{X})}{P(\mathbf{X})}$$

Bringing formal time dependence into the picture, at a given time m:

$$P(\mathbf{y}(m)|\mathbf{X}(m)) = \frac{P(\mathbf{y}(m), \mathbf{X}(m))}{P(\mathbf{X}(m))} \tag{3.2}$$

Because this is a Markov process, the probability of being in a state at a given time is only dependent on the state at the previous time:

$$P(\mathbf{X}(m)) = P(\mathbf{x}(m), \mathbf{X}(m-1))$$

As a result, when the numerator and denominator of equation (3.2) are divided by $P(\mathbf{X}(m-1)$, $P(\mathbf{y}(m)|\mathbf{X}(m))$ can be simplified to:

$$P(\mathbf{y}(m)|\mathbf{X}(m)) = exp[CMI(\mathbf{y}(m), \mathbf{x}(m)|\mathbf{X}(m-1))]P(\mathbf{y}(m)|\mathbf{X}(m-1)) \tag{3.3}$$

where the conditional mutual information between the model state and observations at time m, conditioned on previous observations, $\mathbf{X}(m-1)$, is given by [18]:

$$CMI(\mathbf{y}(m), \mathbf{x}(m)|\mathbf{X}(m-1)) = log\left\{\frac{P(\mathbf{y}(m), \mathbf{x}(m)|\mathbf{X}(m-1))}{P(\mathbf{x}(m)|\mathbf{X}(m-1))P(\mathbf{y}(m)|\mathbf{X}(m-1))}\right\}$$

Now, for t_2 between t_1 and t_3, for Markov processes the Chapman Kolmogorov relation gives[25]:

$$P_{1|1}(y_3(t_3)|y_1(t_1)) = \int P_{1|1}(y_3(t_3)|y_2(t_2))P_{1|1}(y_2(t_2)|y_1(t_1))dy_2$$

which for our D-dimensional model phase space becomes:

$$P(\mathbf{y}(m)|\mathbf{X}(m-1)) = \int d^D\mathbf{y}(m-1)P(\mathbf{y}(m)|\mathbf{y}(m-1))P(\mathbf{y}(m-1)|\mathbf{X}(m-1))$$

Substitution of this relation back into (3.3), and continuing the dynamics back to time t_0 now gives for $P(\mathbf{y}|\mathbf{X})$ [1]:

$$\begin{aligned} P(\mathbf{y}(m)|\mathbf{X}(m)) &= \int d\mathbf{Y} \exp\Big\{\sum_{n=0}^{m} CMI(\mathbf{y}(n),\mathbf{x}(n)|\mathbf{X}(n-1)) \\ &\quad + \sum_{n=0}^{m-1} \log[P(\mathbf{y}(n+1)|\mathbf{y}(n)] + \log[P(\mathbf{y}(0))]\Big\}, \end{aligned} \tag{3.4}$$

which is in the form (3.1) for the action, A_0, defined by:

$$\begin{aligned} A_0(\mathbf{Y},\mathbf{X}) &= -\Big\{\sum_{n=0}^{m} CMI(\mathbf{y}(n),\mathbf{x}(n)|\mathbf{X}(n-1)) \\ &\quad + \sum_{n=0}^{m-1} \log[P(\mathbf{y}(n+1)|\mathbf{y}(n)] + \log[P(\mathbf{y}(0))]\Big\}. \end{aligned} \tag{3.5}$$

3.2.3 Approximation

The two terms for the action, (3.5), are easily separated as a term related to noise in the measurements, $\mathbf{X}$, and a term related to model noise, $\mathbf{Y}$. These terms are treated separately, in order to simplify the sum.

Measurement

As previously discussed, the noise in each measurement is assumed to be independent of the noise at every other measurement. As a consequence, the probability distributions in the conditional mutual information, $CMI(\mathbf{y}(n),\mathbf{x}(n)|\mathbf{X}(n-1))$ are not conditional on the previous state of the system, so the CMI is equal to:

$$\begin{aligned} CMI(\mathbf{y}(n),\mathbf{x}(n)|\mathbf{X}(n-1)) &= \log\left[\frac{P(\mathbf{y}(n),\mathbf{x}(n))}{P(\mathbf{y}(n))\,P(\mathbf{y}(n))}\right] \\ &= \log\left[\frac{P(\mathbf{x}(n)|\mathbf{y}(n)))}{P(\mathbf{x}(n))}\right]. \end{aligned}$$

If the measurement noise is additive and Gaussian, then

$$\sum_{n=0}^{m} CMI(\mathbf{y}(n),\mathbf{x}(n)|\mathbf{X}(n-1)) = -\frac{1}{2}\sum_{n=0}^{m}\sum_{l=1}^{L} R_{meas,l}(y_l(n)-x_l(n))^2, \tag{3.6}$$

where $R_{meas,l}$ is related to the inverse of the Gaussian noise standard deviation, and is generally different for each measured state variable. With the addition of the R_{meas} factor, this term is equivalent to the least squared error term encountered in Chapter 2, prior to addition of a coupling term.

Model

The model error term,

$$\sum_{n=0}^{m-1} \log[P(\mathbf{y}(n+1)|\mathbf{y}(n)] + \log[P(\mathbf{y}(0))]$$

includes a term related to the initial state of the system, as well as conditional probabilities of the state of the system based on each previous state. In a temporally discretized system, this is equivalent to the discussion of errors related to the discretization scheme, as well as any environmental fluctuations, and can be expressed by:

$$\sum_{n=0}^{m-1} \log[P(\mathbf{y}(n+1)|\mathbf{y}(n))] = -\frac{1}{2}\sum_{n=0}^{m-1}\sum_{a=1}^{D} R_{model,a}(y_a(n+1) - f_a(\mathbf{y}(n),\mathbf{p}))^2 \quad (3.7)$$

where $R_{model,a}$ is, similar to $R_{meas,l}$, related to the inverse of the Gaussian noise standard deviation in the model dynamics, and f_a is a shorthand designation for the discretized integration map.

Total Action Approximation

Combining these two approximations, gives

$$\begin{aligned} A_0(\mathbf{Y}|\mathbf{X}(m)) &= \frac{1}{2}\sum_{n=0}^{m}\sum_{l=1}^{L} R_{meas,l}(y_l(n) - x_l(n))^2 \quad (3.8) \\ &\quad + \frac{1}{2}\sum_{n=0}^{m-1}\sum_{a=1}^{D} R_{model,a}(y_a(n+1) - f_a(\mathbf{y}(n),\mathbf{p}))^2 - \log[P(\mathbf{x}(0)]. \end{aligned}$$

3.3 Minimizing the Action

The goal of this technique, called the path integral formulation of parameter estimation, is to find a conditional probability distribution for the state of a system,

based on measurements of a subset of the state variables. With the approximation for A_0, this conditional probability is calculated from:

$$P(\mathbf{Y}|\mathbf{X}(m)) = \int \sum_{n=0}^{m-1} d^D y(n) exp[-A_0(\mathbf{Y}|\mathbf{X}(m))] \tag{3.9}$$

Even with the approximations to simplify the action, A_0, this D-dimensional integral is generally very difficult to compute. One approach is to use a Monte Carlo method, as discussed in [42], to sample many potential paths for the state of the system, thus estimating the distribution $exp[-A_0(\mathbf{Y}|\mathbf{X}(m))]$.

Another method is to use a saddle path approximation to the action, which leads to standard perturbation theory[55]. In this method, the action is expanded about a stationary point $\mathbf{S}$, assuming fixed data:

$$\begin{aligned} A_0(\mathbf{Y}) &= A_0(\mathbf{S}) + \frac{1}{2}(\mathbf{Y}-\mathbf{S})^T A_0''(\mathbf{S})(\mathbf{Y}-\mathbf{S}) + ... \\ \frac{\partial A_0(\mathbf{Y})}{\partial \mathbf{Y}}|_{\mathbf{Y}=\mathbf{S}} &= 0. \end{aligned} \tag{3.10}$$

This stationary path approximation to the path integral gives the same numerical optimization problem described in Chapter 2 for the case of deterministic dynamics. In this case, only the first term of $A_0(\mathbf{Y})$ is used, and the optimization consists of minimizing the error between model and data, subject to the deterministic dynamics of the system as constraints.

Alternatively, this problem can be posed as an unconstrained minimization problem, with the form of the action, (3.8) as the cost function, so that the dynamics of the system are part of the cost function as well. This variation more easily allows for numerical errors in the model, and is especially suited for producing good starting paths for use in Monte Carlo techniques.

Chapter 4

Neuroscience Models

Neuroscience is a broad field with many interesting experimental systems that benefit from utilization of parameter estimation techniques. As an introduction to the types of systems involved, as well as show how powerful the methods in Chapters 2 and 3 can be, we will first discuss the theory of gating models for single neurons.

4.1 Single Neuron Gating Models

Many textbooks, for instance [14], [24] and [26], give a detailed examination of experiments that have led to the current understanding of how neurons generate and propagate information via voltage pulses. Here, we will give a brief summary.

A simple model of a neuron is as a point source, surrounded by a semi-permeable membrane. This membrane is the boundary between the intracellular (inner) and extracellular (outer) regions of the cell, which have different electrical potentials. The difference in these potentials is called the membrane potential. The membrane acts as an insulator to separate charge, i.e., a capacitance.

The cell membrane is made up of protein molecules, some of which form ionic channels, which allow ions to move between the intracellular and extracellular regions. This gives rise to ionic currents, characterized by some resistance. When these currents are in dynamic equilibrium, the cell's potential does not change - this equilibrium voltage is called the resting potential, V_{rest}. A sketch of this simple

electrical circuit is shown in figure (4.1).

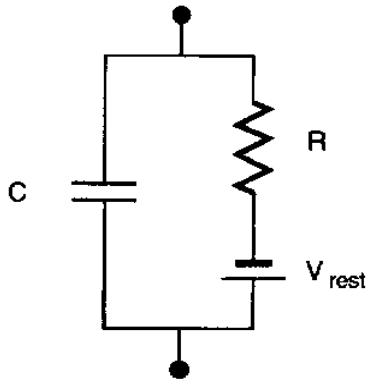

Figure 4.1: Simple RC circuit representation of a cell membrane. A resting potential acts as a battery, and the resistance relates to voltage-independent ionic channels in the membrane.

This simple RC circuit representation does not model real neurons very well. The differential equation for an RC circuit:

$$C\frac{dV_m(t)}{dt} + \frac{V_m(t) - V_{rest}}{R} = I_{inj}(t)$$

can be solved in closed form when the injected current, $I_{inj}(t)$ is a constant, I_0. The solution to this differential equation is:

$$V_m(t) = RI_0(1 - e^{-t/RC}) + V_{rest}$$

which will settle to an equilibrium voltage from any starting conditions, thus not depicting the dynamical range of real neurons. To illustrate this range, more complexity than the simple RC circuit model is necessary.

4.1.1 Hodgkin-Huxley

A fundamental feature of cells in the nervous system is the production of voltage pulses, also known as action potentials or spikes. These spikes originate at the cell body, or soma, and propagate down the dendritic tree, axon, and on to other neurons via synapses. The propagation of these spikes is generally described

by partial differential equations that take into account the spatial structure of the cell; this spatial dependence is not necessary to explain the spike characteristics themselves.

In their seminal paper [23], Hodgkin and Huxley described dynamics that can reproduce the observed initiation and propagation of voltage pulses in the giant axon of the squid. They kept the general RC circuit idea, but added individual ionic currents for sodium and potassium that are voltage dependent, as in figure (4.2). By Kirchhoff's Law, the total membrane current is the sum of these ionic currents and the capacitive current, and each ionic current is described by Ohm's law:

$$I_i(t) \quad = \quad G_i(V(t), t)(V(t) - E_i)$$

where the ionic conductance is voltage and time dependent.

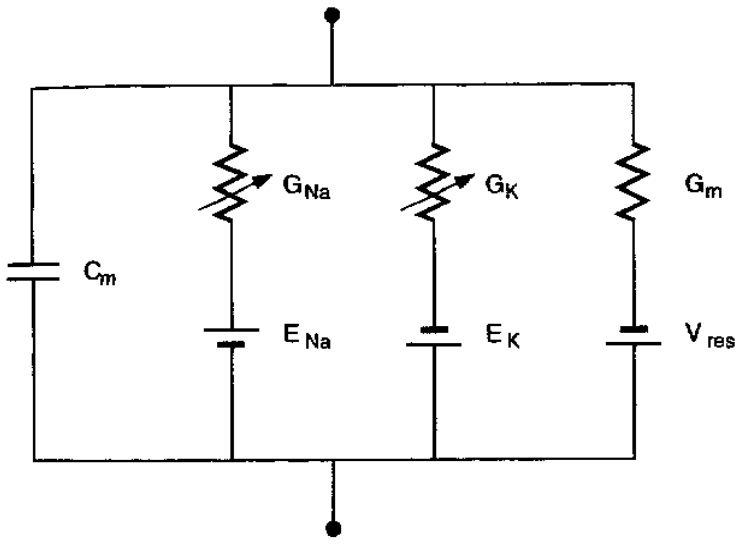

Figure 4.2: RC circuit representation of the Hodgkin-Huxley model

Potassium current

In their work, Hodgkin and Huxley blocked various ionic currents, and studied two major classes of currents: delayed rectifier K^+ and fast Na^+ conductances.

The delayed rectifier K^+ conductance is a persistent conductance with the current modeled as:

$$I_K = \tilde{G}_K n^4 (V - E_K) \tag{4.1}$$

where $\tilde{G}$ is the maximal conductance and E_K is the potassium reversal potential (battery). The gating variable, n, can be thought of as the probability of whether the ion channel is open, and is a non-dimensional number between 0 and 1. Similar numbers are used to describe the fraction of the conductance that is open in other currents; Hodgkin and Huxley described these numbers as gating particles with two states, open and closed, with a transition described by first order kinetics. For example:

$$n \underset{\alpha_n}{\overset{\beta_n}{\rightleftharpoons}} 1 - n$$

Fitzhugh first described the Markovian kinetic model of the Hodgkin-Huxley four-gate, two-state (open and closed) system as a five-state system for the K^+ particles:[20]

$$n_0 \underset{\beta_n}{\overset{4\alpha_n}{\rightleftharpoons}} n_1 \underset{2\beta_n}{\overset{3\alpha_n}{\rightleftharpoons}} n_2 \underset{3\beta_n}{\overset{2\alpha_n}{\rightleftharpoons}} n_3 \underset{4\beta_n}{\overset{\alpha_n}{\rightleftharpoons}} n_4$$

Here, the α's and β's are opening and closing rates of the ion channels, and reflect the fact that the gates are considered identical. For instance, for K^+, the n_4 state is the "open" state, and the closing rate is $4\beta_n$ since the gate will close if any of the four identical gates close. These dynamics form the basis for many theoretical papers that explore the limitations of the Hodgkin-Huxley formulation.

Mathematically, these kinetics correspond to the differential equation,

$$\frac{dn}{dt} = \alpha_n(\mathrm{V}_m)(1 - \mathrm{n}) - \beta_n(\mathrm{V}_m)\mathrm{n}$$

where Hodgkin and Huxley approximated the voltage dependencies of the rate constants to be:

$$\begin{aligned} \alpha_n(V_m) &= \frac{(10 - V_m)}{100(e^{(10-V_m)/10} - 1)} \\ \beta_n(V_m) &= 0.125 e^{-V_m/80} \end{aligned}$$

Sodium current

Similar to the potassium current, Hodgkin and Huxley modeled a fast Na^+ conductance with gating particles as open and closed states, but the dynamics of this current necessitates two types of particles: a sodium activation particle m and an inactivation particle h.

$$I_{Na} = \tilde{G}_{Na} m^3 h (V - E_{Na}) \tag{4.2}$$

Similar to the potassium current, the gating variables correspond to the kinetics:

$$\begin{aligned} \frac{dm}{dt} &= \alpha_m(\mathrm{V}_m)(1 - \mathrm{m}) - \beta_m(\mathrm{V}_m)\mathrm{m} \\ \frac{dh}{dt} &= \alpha_h(\mathrm{V}_m)(1 - \mathrm{h}) - \beta_h(\mathrm{V}_m)\mathrm{h} \end{aligned}$$

with approximated rate constant voltage dependencies,

$$\begin{aligned} \alpha_m(V_m) &= \frac{(25 - V_m)}{10(e^{(25-V_m)/10} - 1)} \\ \beta_m(V_m) &= 4e^{-V_m/18} \\ \alpha_h(V_m) &= 0.07e^{-V_m/20} \\ \beta_h(V_m) &= \frac{1}{e^{(30-V)/10} + 1} \end{aligned}$$

The differential equations for the gating variables can be expressed in an alternate form:

$$\frac{dx}{dt} = \frac{x_\infty - x}{\tau_x}$$

where τ_x is a rate constant

$$\tau_x = \frac{1}{\alpha_x + \beta_x}$$

and x_∞ is the steady state value for the gating variable,

$$x_\infty = \frac{\alpha_x}{\alpha_x + \beta_x}$$

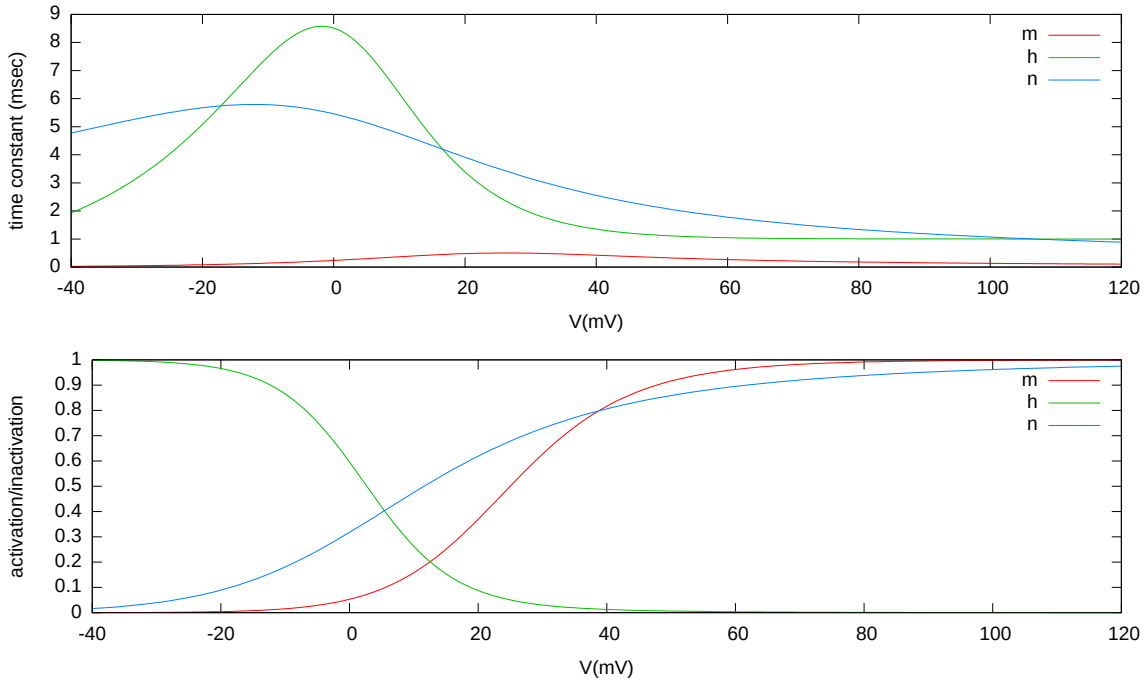

Figure 4.3: Time constants and steady state activation/inactivation of the Hodgkin-Huxley gating particles.

Figure (4.3) shows the time constants and steady state values for Hodgkin-Huxley model gating variables as a function of the membrane voltage.

Fitzhugh's [20] Markovian kinetic description of the sodium channel is an eight-state scheme for the Na^+ particles:

$$\begin{array}{ccccc} m_0h_1 & \underset{\beta_m}{\overset{3\alpha_m}{\rightleftharpoons}} & m_1h_1 & \underset{2\beta_m}{\overset{2\alpha_m}{\rightleftharpoons}} & m_2h_1 & \underset{3\beta_m}{\overset{\alpha_m}{\rightleftharpoons}} & m_3h_1 \\ \alpha_h \uparrow\downarrow \beta_h & & \alpha_h \uparrow\downarrow \beta_h & & \alpha_h \uparrow\downarrow \beta_h & & \alpha_h \uparrow\downarrow \beta_h \\ m_0h_0 & \underset{\beta_m}{\overset{3\alpha_m}{\rightleftharpoons}} & m_1h_0 & \underset{2\beta_m}{\overset{2\alpha_m}{\rightleftharpoons}} & m_2h_0 & \underset{3\beta_m}{\overset{\alpha_m}{\rightleftharpoons}} & m_3h_0 \end{array}$$

For instance, the m_3h_1 state is the open state, and will change to a closed state if any of the three activation gates or the one inactivation gates are closed, with rates of $3\beta_m$ and β_h, respectively.

Complete Model

In addition to the sodium and potassium currents, the membrane includes a leak current, with a voltage independent leak conductance G_m. If a current is

injected into the cell, then the complete Hodgkin-Huxley model is:

$$\begin{aligned}
\frac{dV_m}{dt} &= \frac{1}{\mathrm{C}_m}\left[\tilde{G}_{Na}\mathrm{m}^3\mathrm{h}(E_{Na}-\mathrm{V}_m)+\tilde{G}_K\mathrm{n}^4(E_K-\mathrm{V}_m)+\tilde{G}_m(V_{rest}-\mathrm{V}_m)+I_{inj}\right]\\
\frac{dn}{dt} &= \alpha_n(\mathrm{V}_m)(1-\mathrm{n})-\beta_n(\mathrm{V}_m)\mathrm{n}\\
\frac{dm}{dt} &= \alpha_m(\mathrm{V}_m)(1-\mathrm{m})-\beta_m(\mathrm{V}_m)\mathrm{m}\\
\frac{dh}{dt} &= \alpha_h(\mathrm{V}_m)(1-\mathrm{h})-\beta_h(\mathrm{V}_m)\mathrm{h}
\end{aligned}$$

with the α and β kinetics given previously.

In this treatment, the n, m, and h probabilities are macroscopic averages, and the behavior of an individual ion channel is irrelevant. Treating the ion channels as a macroscopic average, as in the Hodgkin-Huxley model, is appropriate for most physical systems, since the number of excitable channels for an axon's spike initiation is estimated to be large (tens of thousands).

Given initial conditions for V, n, m, and h, the Hodgkin-Huxley equations give a fully deterministic and continuous model to describe the dynamics of neural membranes. Using the giant squid axon model parameters for the maximum conductances and reversal potentials, this model has a single fixed point and does not exhibit spiking in the absence of an applied current ($I_{inj} = 0$). Above a threshold current, the membrane exhibits periodic spiking, indicative of a stable limit cycle.

A typical plot of the Hodgkin-Huxley state variables that result from a short current pulse is shown in Figure (4.4). Note that all the gating variables return to their steady state values, in the absence of the current stimulus. With a long current pulse, the neuron will exhibit periodic spiking.

4.1.2 Morris-Lecar

We would like to test the dynamic parameter estimation technique on problems of the Hodgkin-Huxley type, but first we examine a simpler model that generates a similar spiking response. The Hodgkin-Huxley framework has been shown to give remarkable similarity to experimental data[34, 9, 27], but its mathematical complexity is computational challenging for large networks of neurons.

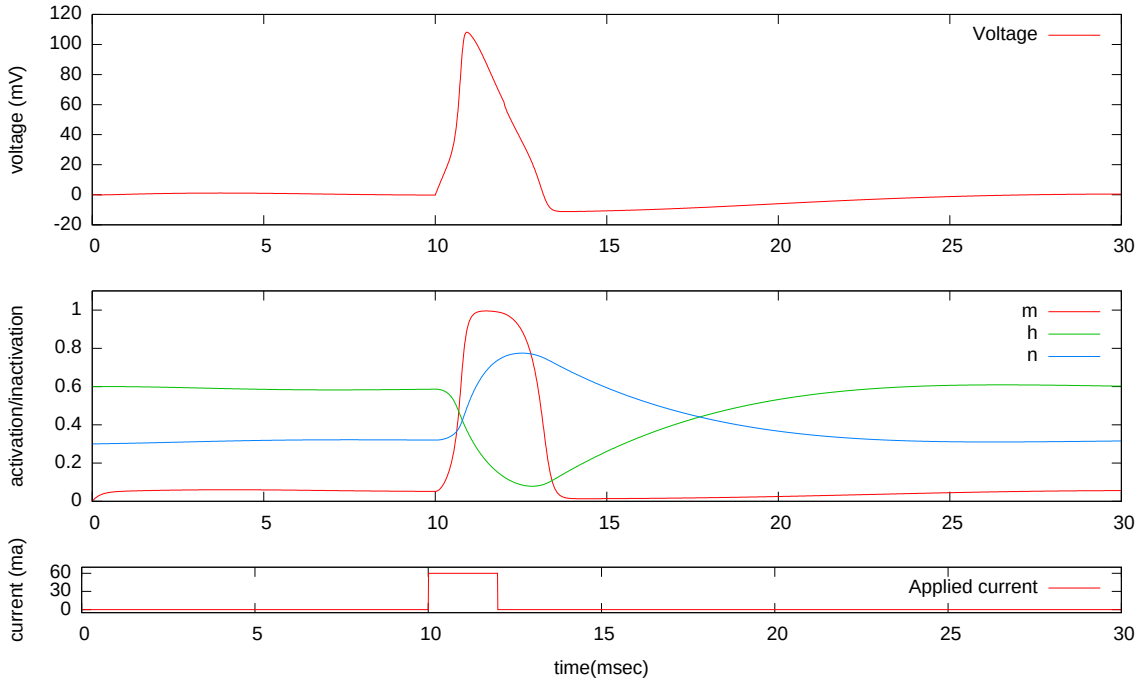

Figure 4.4: Hodgkin-Huxley voltage spiking and gating dynamics in response to a step current injection.

As a result, many attempts have been made to reduce the number of differential equations. Two of the more accomplished reduced models that still exhibit the qualitative spiking features of the Hodgkin-Huxley equations are the FitzHugh-Naguma [19, 21, 36] and Morris-Lecar [35] models. Here, we will examine the Morris-Lecar model.

This model was developed to describe barnacle muscle fibers subject to constant current inputs, and consists of a set of coupled differential equations which include two ionic currents. In contrast to the Hodgkin-Huxley model, no sodium current is present; instead the two currents are an outward potassium current and an inward calcium current. In this case, the calcium is assumed to be in equilibrium, and its dynamics can be ignored. The coupled equations for the Morris-Lecar model are:

$$\begin{aligned} C_m \frac{dV_m}{dt} &= -I_{ion}(V_m, w) + I(t) \\ \frac{dw}{dt} &= \frac{w_\infty(V_n) - w}{\tau_w(V_m)} \end{aligned}$$

Here, V_m is the membrane voltage, w is the potassium activation variable, and the ionic current is a combination of potassium, calcium, and leak currents:

$$I_{ion} = \tilde{G}_{Ca} m_\infty (V_m - E_{Ca}) + \tilde{G}_K w (V_m - E_K) + \tilde{G}_m (V_m - V_{rest})$$

The calcium current is always at equilibrium, with

$$m_\infty(V_m) = 0.5\left(1 + tanh\frac{V_m + 1}{15}\right)$$

and the potassium dynamics is governed by time constant:

$$\tau_w(V_m) = \frac{5}{coshV_m/60}$$

and steady state activation:

$$w_\infty(V_m) = 0.5\left(1 + tanh\frac{V_m}{30}\right)$$

An example of the voltage spiking seen in the Morris-Lecar model in response to a step current is shown in Figure (4.5).

4.2 Parameter Estimation Example

We will use the Morris-Lecar model to demonstrate the dynamic state and parameter estimation procedure for neuron models. From Chapter 2, recall that the goal of the method is to determine unmeasured parameters and states in a model based on the measurement of a subset of the state variables. In this case, we will measure the membrane voltage from a Morris-Lecar neuron, and determine the reversal potentials, maximal conductances, and coefficients within the calcium and potassium dynamics.

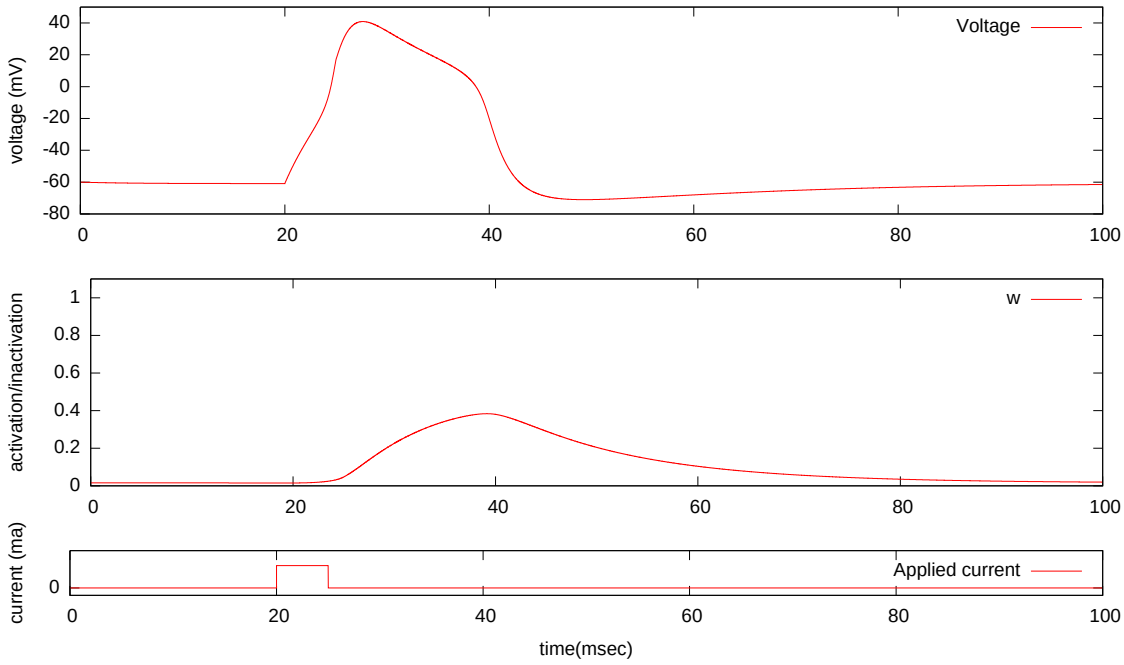

Figure 4.5: Morris-Lecar voltage spiking and gating dynamics in response to a step current injection.

4.2.1 Morris-Lecar

The general optimization problem described in Chapter 2 is modified to implement the Morris-Lecar model dynamics. The procedure is the same as described in Section 2.5. The problem is coded into Fortran code that calls optimization libraries for the software SNOPT [22],a general-purpose system for constrained optimization. This software is suitable for general nonlinear programs, and minimizes a nonlinear function subject to bounds on the variables and sparse linear or nonlinear constraints.

SNOPT uses a sequential quadratic programming (SQP) algorithm. Search directions are obtained from QP subproblems that minimize a quadratic model of the Lagrangian function subject to linearized constraints, which requires knowledge of the analytical Jacobian matrix of the constraint equations. In our problem, the Simpson integration and Hermite Polynomial interpolation for each state variable at each time point are separate constraints. Each state variable at each time point,

as well as all parameters are unknowns in the problem.

The Jacobian matrix consists of derivatives of each of the constraint equations with respect to each of the unknowns. The bulk of the Fortran optimization code consists of setting up the objective function, constraint equations, and Jacobian matrix correctly. To easily differentiate between state variables and parameters, we introduce a new notation for the Morris-Lecar equations.

The dynamical variables in the model are the membrane voltage $y_1(t)$ and the K^+ activation variable $y_2(t)$. Parameters are chosen as well as an $I_{\text{app}}(t)$ and the following ordinary differential equations are solved for the data $V_m(t) = x_1(t)$ and $w(t) = x_2(t)$.

$$\begin{aligned}
\frac{dx_1(t)}{dt} &= p_{12}\Big(p_9 m_\infty(x_1(t))(p_5 - x_1(t)) + \\
&\quad p_{10}x_2(t)(p_6 - x_1(t)) + p_{11}(p_7 - x_1(t)) + I_{\text{app}}(t)\Big) \\
\frac{dx_2(t)}{dt} &= \frac{w_\infty(x_1(t)) - x_2(t)}{\tau_w(x_1(t))}.
\end{aligned}$$

where

$$\begin{aligned}
m_\infty(V) &= \frac{1}{2}(1 + \tanh(\frac{(V - p_1)}{p_2})) \\
w_\infty(V) &= \frac{1}{2}(1 + \tanh(\frac{(V - p_3)}{p_4})) \\
\tau_w(V) &= p_8/\cosh(\frac{(V - p_3)}{2p_4}).
\end{aligned}$$

The parameters in the "data" source for $x_1(t)$ were selected as $p_1 = -1.2\,\text{mV}$, $p_2 = 18.0\,\text{mV}$, $p_3 = 2\,\text{mV}$, $p_4 = 30\,\text{mV}$, $p_5 = 120\,\text{mV}$, $p_6 = -84\,\text{mV}$, $p_7 = -60\,\text{mV}$, $p_8 = 0.04$, $p_9 = 4.4\text{mS/cm}^2$, $p_{10} = 8\text{mS/cm}^2$, $p_{11} = 2\text{mS/cm}^2$, and $p_{12} = 0.05\text{cm}^2/\mu F$.

The data $x_1(t)$ were coupled into $y_1(t)$ with the control term u(t):

$$\frac{dy_1(t)}{dt} = p_{12}\Big(p_9 m_0(y_1(t))(p_5 - y_1(t)) + p_{10}y_2(t)(p_6 - y_1(t)) + p_{10}y_2(t)(p_6 - y_1(t)) + p_{11}(p_7 - y_1(t)) + I_{\text{app}}(t) + u(t)(x_1(t) - y_1(t))\Big)$$

$$\frac{dy_2(t)}{dt} = \frac{w_0(y_1(t)) - x_2(t)}{\tau(y_1(t))}.$$

We first performed two current clamp "experiments" where $I_{\text{app}}(t)$ was 0 until $t = 50\,\text{ms}$, then I_1 for $50\,ms \leq t \leq 220\,\text{ms}$. We selected $I_1 = 75\mu A/\text{cm}^2$ for one calculation. At this value of the injected current, the Morris-Lecar model has a fixed point or a constant resting voltage. A second current clamp experiment was performed with $I_1 = 115\mu A/\text{cm}^2$, at which current level the ML neuron undergoes periodic limit cycle oscillations.

After this current clamp protocol, we investigated another applied current $I_{\text{app}}(t) = 220(1 - \cos(0.02t)e^{-0.005t})\ \mu A/\text{cm}^2$. The parameters in the ML model should be independent of $I_{\text{app}}(t)$. Table 4.1 shows the results of each injected current protocol on the determination of the parameters in the ML model.

4.2.2 Morris-Lecar Results

As seen in Table 4.1, and Figures 4.7 and 4.6, the dynamic state and parameter estimation technique accurately estimates the unmeasured parameters in this model for a range of applied currents, as well as the unmeasured state variable, w. Importantly, the control term, u(t), is reduced to zero, and the R-value is 1 at every time point.

4.3 Discussion

Even though this two dimensional system does not show chaos, it is highly non-linear with a large number of parameters compared to the Lorenz 1963 system. The fact that the dynamical state and parameter estimation technique works

Table 4.1: Morris Lecar Model: "Data" Parameters and Estimated Parameters. $I_{\text{app}}(t) = 220(1 - \cos(0.02t)e^{-0.005t})\ \frac{\mu A}{\text{cm}^2}$.

Parameter	Data	$I_{\text{app}}(t)$	$I_1 = 75\,\mu A/\text{cm}^2$	$I_1 = 115\,\mu A/\text{cm}^2$
$p_1 = V_1$ mV	-1.2	-1.20002	-1.11	-1.13
$p_2 = V_2$ mV	18.0	17.999	18.33	18.219
$p_3 = V_3$ mV	2.0	2.0008	2.80	2.64
$p_4 = V_4$ mV	30.0	30.0008	31.22	30.93
$p_5 = E_{Ca}$ mV	120.0	120.0011	120.85	120.68
$p_6 = E_K$ mV	-84.0	-83.9999	-84.25	-84.27
$p_7 = E_L$ mV	-60.0	-59.9998	-59.94	-59.93
$p_8 = \phi\ (ms)^{-1}$	0.04	0.040000	0.04113	0.0407
$p_9 = g_{Ca}\ mS$	4.4	4.3998	4.434	4.411
$p_{10} = g_K$ mS/cm^2	8.0	8.0000	8.07	8.05
$p_{11} = g_L$ mS/cm^2	2.0	1.99997	2.03	2.02
$p_{12} = 1/C_M$ cm$^2/\mu F$	0.05	0.04999	Fixed at 0.05	Fixed at 0.05

so well is an important stepping stone toward more complicated neuron models, such as the Hodgkin-Huxley model, as well as multiple-compartment and network models.

However, the Fortran coding to implement the Morris-Lecar model took two months, due to troubleshooting the correct form of the complicated derivatives necessary for the analytical Jacobian, and any more complicated models could take comparable or significantly more time to debug. Since we wish to expand this parameter estimation technique to larger network models, a new way of generating problem files is necessary.

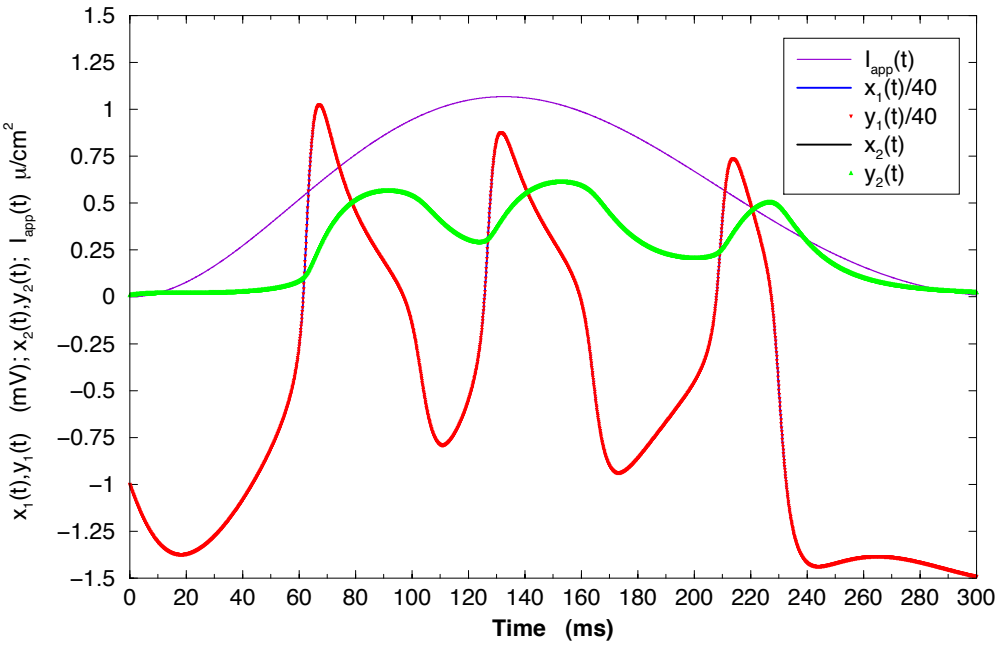

Figure 4.6: Morris-Lecar Unobserved State Variable Estimation for applied current pulse when of $I_{\text{app}}(t) = 220(1 - \cos(0.02t)e^{-0.005t})\ \frac{\mu A}{\text{cm}^2}$.

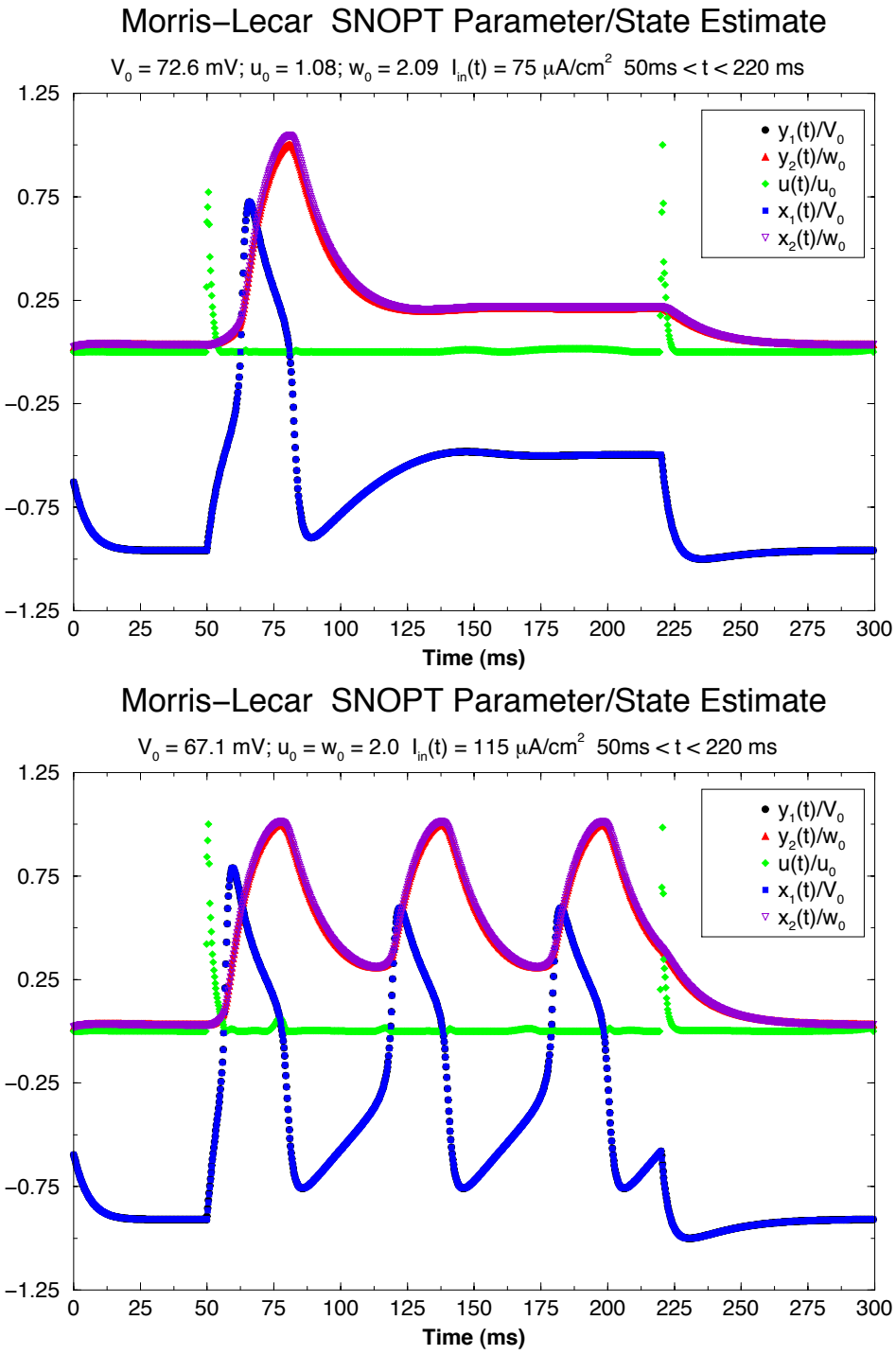

Figure 4.7: Morris-Lecar Unobserved State Variable Estimation for constant applied current pulse of 75 and 115 $\frac{\mu A}{\text{cm}^2}$.

Chapter 5

Python Scripting

The basic structure of a dynamic parameter estimation problem is similar for all vector fields - the Morris-Lecar example was one of the toy models used as a proof of concept for the method. The main goal of this method of parameter and state estimation is to determine unmeasured parameters and states from real, physical systems.

Each new problem has a unique model associated with it, so a new optimization instance must be constructed. For toy models such as the Lorenz 1963 model, the vector field and Jacobian matrix can be readily calculated and input by hand, but this quickly becomes cumbersome for complex models of real systems, such as the Morris-Lecar and Hodgkin-Huxley models. To facilitate the use of the dynamical parameter estimation method to a new problem, I have used the Python programming language to develop scripts that set up the problem in the correct format for use with readily available optimization software. These scripts take a simple text file formulation of a parameter estimation vector field and output correctly formatted and linked C^{++} files for use with the widely available IPOPT software libraries [54].

In order to implement this method, I use the Python programming language to develop scripts that generate C^{++} files to set-up the problem in the correct format. These scripts read two text files which define the vector field of the dynamical system, as well as the feasibility space of the optimization. Using a time-series observation of one (or more) of the state variables, the generated

C^{++} code determines the unmeasured parameters of the system, $\mathbf{q}$, as well as the unobserved state variables using the dynamical parameter estimation technique. In addition, I have extended the use of these text files with Python scripting to solve a different optimization problem, based on the path integral formulation in Chapter 3.

5.1 Overview

Python is a multi-purpose programming language that permits both object-oriented programming and structured programming. Python can be used as a scripting language or a full-fledged programming language, and running an IPOPT or SNOPT optimization can be done with appropriate use of Python modules that interact with C^{++} and Fortran libraries. Python is a higher-level language than C^{++} or Fortran; as a result program development is generally easier, but the execution speed is slower. For the dynamic parameter estimation optimization problems, which currently can consist of tens of thousands of variables and constraints, with the potential for many more, the program execution speed in C^{++} (native language of IPOPT) or Fortran (native language of SNOPT) can be significantly faster than in Python.

The Python scripts define a problem with two distinct text files. The first text file, called 'equations.txt', defines the vector field of the model. The number of equations, parameters, and controls are specified, along with the variable names, model vector field and objective function. As an example, for the Hodgkin-Huxley model, there are four dynamical variables which correspond with the four differential equations of the model, three parameters to be determined, and one control variable. Once this text file is fully defined, the Python module, 'makecode.py' sets up the C^{++} files necessary to run the optimization. The 'equations.txt' file for the Hodgkin-Huxley dynamic parameter estimation problem is shown in Figure 5.1.

A second text file, called 'specs.txt' includes problem parameters that the compiled C^{++} executable program loads to run a particular instance of a problem.

'specs.txt' includes the size of the data file (number of data points), sampling frequency of the data, names of the data files needed by the executable, as well as the variable bounds and initial conditions (guess) for the optimization. The 'specs.txt' file for the Hodgkin-Huxley dynamic parameter estimation problem is shown in Figure 5.2.

5.2 Python Scripts

5.2.1 Discretize.py

The module 'makecode.py' consists of several separate Python scripts, that run sequentially. One of the strengths of the Python language relative to C^{++} and Fortran is the ease of string manipulations. The module that uses this string functionality to set up the optimization problem is 'discretize.py'. 'discretize.py' uses the SymPy package, a Python library for symbolic mathematics[10] for this purpose. The state variable, parameter, and control names, as well as the vector field and objective function for the problem are imported from 'equations.txt' and converted into symbolic equations for use in SymPy. These symbolic equations are then transformed into a discretized integration according to Simpson's Rule with polynomial interpolation described previously.

This discretized form of the vector field is then used to symbolically calculate the Jacobian and Hessian matrices, using SymPy to take first and second derivatives of the vector field with respect to all the state variables, parameters, and controls. For the Simpson's Rule discretization choice, care is taken to keep track of whether state variable and control derivatives are taken with respect to the variable at the current time, next time, or mid-point time, to ensure proper placement within the Jacobian and Hessian structures. The result is symbolic arrays for the vector field, Jacobian, and Hessian, that include non-zero entries only, since these matrix elements will be entered into the optimization in a standard sparse matrix formulation that only needs the row, column, and value information for non-zero entries. Five separate arrays from 'discretize.py' are needed in the IPOPT optimization:

- Objective function
- Constraints
- Gradient of the Objective Function
- Jacobian of the Constraints
- Hessian of the Objective Function and Constraints

The symbolic strings in these arrays are converted into proper C^{++} format (e.g., ** changed to pow function) and stored for the next part of the process, 'makecode.py' itself.

5.2.2 Makecode.py

A C^{++} IPOPT optimization program consists of a main program that initiates a new problem class and calls the optimization process. The problem class is defined in another file, with an appropriate header file. For our purposes, the main program is standard across different vector fields; the details of the vector fields are contained within the other files, which we call problemname_nlp.cpp and subsequent header file, problemname_nlp.hpp. The module 'makecode.py' takes the information outputted from 'discretize.py' and writes these C^{++} files for a particular problem. The executable C^{++} program, once compiled, loads the information in specs.txt in order to run the program. In this way, a given vector field can be sampled over various data sets of differing lengths, without re-compilation of the program.

An example of one of the routines that 'makecode.py' generates is the 'Eval-g' function, which returns the value of the constraints. These constraints have been stored in an array in symbolic discretized form by 'discretize.py'. The code generation for this function consists of setting up a loop over all time points, so that each discretized constraint is defined at each time point. Each symbolic variable from 'discretize.py' is equated to the proper term of a C^{++} array that contains all optimization variables. The 'Eval-g' function for the Hodgkin-Huxley problem defined by the Figure 5.1 'equations.txt' file is shown as Figure 5.3. This

function is rather straightforward; for the four dynamical variable model, 46 lines of C^{++} code is generated to define these as constraints. The functions that define the Jacobian and Hessian of the vector field are significantly more complicated, since both functions require detailed row and column information that defines the constraint and partial derivative that is being taken.

The file that sets one problem apart from another is the problem class file. The problem class file consists of a constructor and nine virtual methods to be implemented, which are all declared in the header file. The methods are implemented by 'makecode.py' as follows:

- Constructor: Imports the information in specs.txt for use by the nine virtual methods. This includes bounds for all state variables, parameters, and controls, as well as an initial starting guess for each of these. Also, the experimental data time series are loaded into arrays that can be accessed by the other processes.

- Get_nlp_info: Contains the number of equations, constraints, and non-zero Jacobian and Hessian entries. These are determined in discretize.py.

- Get_bounds_info: Takes information provided by specs.txt to give bounds on all variables in the problem, defining the feasibility space of the optimization. Also sets all constraint equations as equality constraints by default, although an option with the specs.txt construct allows for relaxation to inequality constraints within some range.

- Get_starting_point: Takes information provided by specs.txt to give the initial starting point for the optimization, within the bounds specified in Get_bounds_info. The starting point for state variables can be given as a constant for all discretized time points, or specified for each time point of the optimization. For complicated problems, a starting point close to the expected solution may give a better optimized solution.

- Eval_f: Defines, in terms of the IPOPT optimization variables, the symbolic variables and objective function from discretize.py. Sets up a loop to calculate

the objective function for the optimization by adding contributions from each time step.

- Eval_grad_f: Defines, in terms of the IPOPT optimization variables, the symbolic variables and gradient of the objective function from discretize.py. Sets up a loop to calculate the gradient of the objective function for the optimization at each time step.

- Eval_g: Defines, in terms of the IPOPT optimization variables, the symbolic variables and constraint functions from discretize.py. These symbolic constraint functions are just the equations given in equations.txt transformed into the discretized integration form. Sets up a loop over all data points to calculate the constraint functions for the optimization at each time step.

- Eval_jac_g: Defines, in terms of the IPOPT optimization variables, the symbolic variables and Jacobian elements from discretize.py. Sets up a loop over all data points to set up the non-zero row/column structure of the Jacobian, and a separate loop to calculate the values of the Jacobian at each point in the structure.

- Eval_h: Defines, in terms of the IPOPT optimization variables, the symbolic variables and Hessian elements from discretize.py. Sets up a loop over all data points to set up the non-zero row/column structure of the Hessian, and a separate loop to calculate the values of the Hessian at each point in the structure. Care is taken to differentiate between Hessian elements that need to be calculated at multiple time points (i.e., the two Hessian derivatives are both with respect to state variables) and elements that occupy only one matrix position (i.e, two derivatives are both with respect to parameters).

- Finalize_solution: Prints to file the final optimization values for the parameters, as well as state variables and controls at all time points.

The total length of generated C^{++} code for each function for the Hodgkin-Huxley example is shown in Table 5.1.

Table 5.1: Length of generated code for various subroutines in Hodgkin-Huxley example.

Routine	Description	Lines
Eval_f	Evaluate Objective	73
Eval_grad_f	Evaluate Objective Gradient	75
Eval_g	Evaluate Constraints	46
Eval_jac_g	Evaluate Jacobian	371
Eval_h	Evaluate Hessian	1178

For large problems, defined in terms of the number of dynamical variables (or number of equations), the generated C^{++} code can be very large, as shown in Table 5.2. Since debugging of such a simple model as Morris-Lecar was already cumbersome, these larger models would be extremely difficult to code without this software.

Table 5.2: Length of generated code for various dynamical parameter estimation problems. The listed problems are the Colpitts oscillator, the Morris-Lecar neuron model[35], the Lorenz 1996 atmospheric model[31, 32], the Hodgkin-Huxley neuron model[23], the lobster lateral pyloric neuron model[38], and the barotropic vorticity ocean circulation model[5]

Model	Type	Vars	Cons	equations.txt	C^{++} lines
Colpitts	Osc	10T+8	7T	37 lines	996
ML	Neur	8T+15	5T	42 lines	1260
Lorenz	Atm	18T+10	12T	39 lines	1394
HH	Neur	12T+28	9T	57 lines	2171
lobster	Neur	44T+86	35T	145 lines	6258
Arakawa	Ocean	384T+195	192T	283 lines	218034

5.2.3 Additional scripts

Discretize.py and makecode.py are the workhorse scripts to produce viable code for dynamical parameter optimization. In order to set problems up for use with specific software (e.g., IPOPT), additional scripts are necessary to fill in the gaps. These scripts are called from within makecode.py, therefore only one command is necessary to set up a problem from a given equations.txt file.

Makecpp.py

Makecpp.py writes the main program that initiates a new problem class and calls the optimization process. As noted before, this program is largely the same across multiple problems, with the significant exception of the name of the problem class.

Makehpp.py

Makehpp.py writes the header file for the problem class, problemname_nlp.hpp. This file defines all the methods used in problemname_nlp.cpp, as well as the parameters and arrays that are introduced in the constructor of problemname_nlp.cpp. This includes the number of variables, parameters, and controls in the problem, arrays to hold the symbolic outputs for the controls, variables, and parameters from discretize.py, arrays to hold the time series of data for the optimization, and arrays to hold bounds and initial conditions given in specs.txt.

Makeopt.py

Makeopt.py writes an options file problemname.opt that the main program references prior to running an optimization. Makeopt.py sets defaults for maximum number of iterations and linear solver choice. Once makecode.py has initialized this options file, it can be easily edited for customization of a particular problem or hardware configuration.

Makemake.py

Makemake.py writes a Makefile that compiles the C^{++} files and links them to the IPOPT libraries. This is the only script that must be changed between software and hardware configurations, based on testing with multiple Macintosh and Linux installations on various CPU architectures, simply because the compilation flags and path to the IPOPT libraries for a given installation must be given correctly. For a given installation of IPOPT, this file must only be modified once, and then can be used for multiple optimization instances on the same CPU. Once makemake.py is configured correctly, the command sequence $*makecode.py* followed by $*make* will produce a properly compiled executable from which to run the optimization.

Correlate.py

Correlate.py calculates the R-value result from the optimization. Recall that the R-value is a check to ensure that the coupling u(t) terms become small as a result of the optimization. The R-value measures the relative contributions of the equation dynamics, $F_1(y_1(t), \mathbf{y}_\perp(t), \mathbf{p})$, and the synchronization term, $u(t)(x_1(t) - y_1(t))$, and is defined as the ratio:

$$R - value \quad = \quad \frac{[F_1(y_1(t), \mathbf{y}_\perp(t), \mathbf{p})]^2}{[F_1(y_1(t), \mathbf{y}_\perp(t), \mathbf{p})]^2 + [u(t)(x_1(t) - y_1(t))]^2}$$

Correlate.py calculates an R-value for each equation with a synchronization term, and outputs to the file Rvalue.dat.

5.2.4 Output Files

The output from running a dynamical control problem will generate five data files.

- **Ipopt.out** gives details about the optimization iterations.
- **Param.dat** lists the estimated parameter values in the order given in equations.txt, one per line.

- **Data.dat** lists the estimated variables and controls at each time step. The format of data.dat is: counter, x1, x2, ..., xn, u1, u2, ..., uk, followed by the y1, ..., yk of the input data.

- **Rvalue.dat** lists the Rvalue at each time point. The Rvalue is a measure of the relative contribution of the model dynamics and control terms in the output, and gives a measure of the quality of the state estimation.

- **Carl.input** outputs the state variables and parameters in the format needed for input into the cudaPIMC code, a Markov chain Monte Carlo path integral code developed in accordance with the methods presented in Chapter 3, see reference [42].

5.3 MinAzero

Chapter 3 discussed a path integral formulation of state and parameter estimation, based on minimizing the action of a dynamical system. The goal of this technique, called the path integral formulation of parameter estimation, is to find a conditional probability distribution for the state of a system, based on measurements of a subset of the state variables. With the approximation for A_0, this conditional probability can calculated from:

$$P(\mathbf{Y}|\mathbf{X}(m)) = \int \sum_{n=0}^{m-1} d^D y(n) exp[-A_0(\mathbf{Y}|\mathbf{X}(m))] \tag{5.1}$$

Recall equation (3.8) for the approximation of the action:

$$\begin{aligned} A_0(\mathbf{Y}|\mathbf{X}(m)) &= \frac{1}{2}\sum_{n=0}^{m}\sum_{l=1}^{L} R_{meas,l}(y_l(n) - x_l(n))^2 \\ &\quad + \frac{1}{2}\sum_{n=0}^{m-1}\sum_{a=1}^{D} R_{model,a}(y_a(n+1) - f_a(\mathbf{y}(n), \mathbf{p}))^2 - \log[P(\mathbf{x}(0)]. \end{aligned} \tag{5.2}$$

This conditional probability function is typically solved using Markov Chain Monte Carlo methods, for which convergence is difficult to define. To help in convergence to the correct probability distribution, the initial path (or paths) should be taken from a distribution that is indicative of the expected distribution.[12]

One potential starting path for this method is the Carl.input file outputted from implementation of the DSPE method. In addition, I developed a suite of python scripts, based on the makecode scripts above, with the goal of generating a "good" initial starting path for input to Markov Chain Monte Carlo software, such as discussed in Reference [42]. Whereas the Markov Chain Monte Carlo method attempts to solve equation (5.1), this software, called 'Minazero.py', sets up an IPOPT optimization problem to minimize equation (5.2) in an unconstrained optimization.

5.3.1 Python Scripts

The minazero scripts mirror the makecode scripts to include the following:

- Minazero.py
- Discazero.py
- MakecppAzero..py
- MakehppAzero.py
- MakeoptAzero.py
- Makemakeazero.py

The latter four scripts above are almost identical to the makecode equivalents; only minAzero.py and discAzero.py are significantly changed, to adjust for inclusion of the dynamical equations as part of the objective function and taking into account the unconstrained structure of the optimization.

As before, the module that uses the Sympy package to set up the optimization problem is 'discAzero.py'. The state variable, parameter, data, and control names (if any), as well as the vector field and least squared error term for the problem are imported from 'equations.txt' and converted into symbolic equations for use in SymPy. These symbolic equations are then transformed into a discretized integration according to Simpson's Rule with polynomial interpolation, similar to 'discretize.py'.

Where the minAzero scripts deviate from the makecode implementation is in how these symbolic equations are handled. In makecode, these equations are kept separate and depicted as constraint equations in the optimization; in minAzero these equations become the second term of equation (5.2), and thus the discretized version must be squared. This is again implemented symbolically using SymPy, and then used to symbolically calculate the objective function gradient and Hessian matrix, using SymPy to take first and second derivatives of the vector field with respect to all the state variables, parameters, and controls. For the Simpson's Rule discretization choice, care is taken to keep track of whether state variable and control derivatives are taken with respect to the variable at the current time, next time, or mid-point time, to ensure proper placement within the objective function gradient (previously the Jacobian of constraints) and Hessian structures.

Now, the result is symbolic arrays for the objective function, objective function gradient, and Hessian, that include only non-zero entries only, since these matrix elements will be entered into the optimization in a standard sparse matrix formulation that only needs the row, column, and value information for non-zero entries. For this unconstrained optimization, no Jacobian is necessary. The Hessian is also quite different from the makecode case; the discretized vector field is squared and thus includes cross terms across different time points which must be accounted for.

The symbolic arrays from 'discAzero.py' are loaded into 'minAzero.py', which generates the C^{++} problem files. The executable C^{++} program, once compiled, loads the information in specs.txt in order to run the program, as before. The main difference in this specs.txt file is the inclusion of the $R_{model,a}$ terms from equation (5.2) that quantify the standard deviation in the noise of each dynamical variable.

5.3.2 Output Files

The output from running a minAzero problem will generate four data files - all those for makecode problems, with the exception of the Rvalue.dat file. The Rvalue.dat file is specific to dynamical state and parameter estimation, and has

no significance to the path integral formulation.

5.4 Discussion

Dynamical parameter estimation has applications in a wide range of fields, and these Python scripts have made the implementation for a new model to be straightforward. The version discussed here uses Simpson's integration rule and the IPOPT solver, but these can be easily substituted. For instance, the integration rule is defined in just a few lines of the discretize.py code, and can be changed out to another rule fairly simply. Use of another optimization solver is slightly more complicated since program syntax varies across solvers, but the general front-end algorithms are similar among optimization software.

More importantly, no language-specific programming knowledge is necessary in order to use these scripts for dynamical parameter estimation. The scripts are written in Python, but are run from the a terminal command line, so no Python-specific knowledge is needed. Optimization software packages typically include interfaces to allow the use of a user's language of choice, but these interfaces may not be as well-supported as the base software, may be inefficient to use, and may significantly slow down the computational time to solve a given problem. By writing code in the native language of the optimization software, these Python scripts require only basic command line skills to solve complicated dynamical parameter estimation problems.

Python is the natural programming language to implement this type of program due to its string handling ability. Higher level programs such as MATLAB and Mathematica are capable of implementing the symbolic differentiation performed by SymPy, and many programming languages can be set to write text to a file (which is what the Python scripts ultimately do), but struggle to seamlessly import information from a text file. These Python scripts would not be necessary if only a single optimization instance needed to be performed on a unique model vector field. The ability to radically change the system being studied by editing just two text files opens the door for dynamical parameter estimation to be easily

implemented across many fields.

```
# Problem Name
HH
# nY.nP,nU,nI
4,22,1,1
# equations
(gNa*mm*mm*mm*hh*(ENa-VV)+gK*nn*nn*nn*nn*(EK-VV)+
             gM*(Erest-VV)+Iinj)/Cm+k1*(Vdata-VV)
(1-mm)*(amV1-VV)*amC/((exp((amV1-VV)*amV3)-1.0))-mm*bmC*exp(-VV*bmV1)
(1-hh)*ahC*exp(-VV*ahV1)-hh*bhC/(exp((bhV1-VV)*bhV2)+1.0)
(1-nn)*(anV2-VV)*anC/((exp((anV2-VV)*anV3)-1.0))-nn*bnC*exp(-VV*bnV1)
# Objective/Cost function
(Vdata-VV)*(Vdata-VV)+k1*k1
# variable names
VV
mm
hh
nn
# parameter names
Cm
gNa
ENa
gK
EK
gM
Erest

...

anV2
anV3
bnC
bnV1
# control, data, stimuli names
k1
Vdata
Iinj
```

Figure 5.1: Sample equations.txt file for the Hodgkin-Huxley dynamical parameter estimation problem

```
# Problem Size
200
# How much data to skip
0
# Time step
0.02
# Data File names - input
hhv.dat
# Data File name - stimuli
hhi.dat
# Boundary & initial conditions
0
# State Variables
-200, 200, 0      # V
0, 1, 0.5         # m
0, 1, 0.5         # h
0, 1, 0.5         # n
# Controls:
0, 100, 0
-1,1,0
# Parameters
0, 2, 1
100, 150, 105
100, 150, 105
20, 50, 45
-25, 0, -20
0.1, 0.5, 0.4
5.0, 15.0, 6.0

...

5.0, 15.0, 10.0
0.05, 1.0, 0.1
0.1, 0.5, 0.125
0.01, 0.02, 0.0125
```

Figure 5.2: Sample specs.txt file for the Hodgkin-Huxley dynamical parameter estimation problem

```
bool HH_NLP::eval_g(Index n,const Number* x,bool new_x,Index m,Number* g)
{
  assert(n == 12*Time+28);
  assert(m == 9*Time);

  for(Index jt=0;jt<Time;jt++) {

     for(Index i=0;i<nY;i++) {
        Xval[i] = x[jt + i*(Time+1)];
        Xvalp1[i] = x[jt + i*(Time+1) + 1];
        Xval2[i] = x[(Time+1)*(nY+2*nU) + jt + i*(Time)];
     } //end for loop

     for(Index i=0;i<nU;i++) {
        K11val[i] = x[jt + nY*(Time+1) + 2*i*(Time+1)];
        K11valp1[i] = x[jt + nY*(Time+1) + 2*i*(Time+1) + 1];
        K11val2[i] = x[(Time+1)*(nY+2*nU) + (nY+2*i)*Time + jt];
        dK11val[i] = x[jt + (nY+2*i+1)*(Time+1)];
        dK11valp1[i] = x[jt + (nY+2*i+1)*(Time+1)+1];
        dK11val2[i] = x[(Time+1)*(nY+2*nU) + (nY+2*i+1)*Time + jt];
     } //end for loop

     Xdval[0] = Vdata[2*jt];
     Xdval2[0] = Vdata[2*jt+1];
     Xdvalp1[0] = Vdata[2*jt+2];
     Ival[0] = Iinj[2*jt];
     Ival2[0] = Iinj[2*jt+1];
     Ivalp1[0] = Iinj[2*jt+2];

     for(Index i=0;i<nP;i++)  Pval[i] = x[(2*Time+1)*(nY+2*nU)+i];

     g[9*jt+0] = Xval[0] + 0.166666666666667*hstep*(K11val[0]*  ...

     ...

     g[9*jt+8] = 0.5*K11val[0] + 0.125*dK11val[0]*hstep + 0.5*  ...

  } //end for loop
  return true;
}
```

Figure 5.3: Sample eval_g C^{++} routine for the Hodgkin-Huxley dynamical parameter estimation problem

Chapter 6

Applications of Scripting

The ability to easily generate optimization instances of complicated vector fields opens up a wide range of problems to study. The Hodgkin-Huxley model example gives some valuable insights into how to use these tools, and paves the way for the study of more complicated systems. Two of these systems are a model of the lateral pyloric neuron in the California spiny lobster, and a simple three-neuron network with dynamic synaptic connections.

6.1 Hodgkin-Huxley Model

As the leading example, recall the Hodgkin-Huxley model defined in Chapter 4 by:

$$\begin{aligned}
\frac{dV_m}{dt} &= \frac{1}{\mathrm{C}_m}\left[\tilde{G}_{Na}\mathrm{m}^3\mathrm{h}(E_{Na}-\mathrm{V}_m)+\tilde{G}_K\mathrm{n}^4(E_K-\mathrm{V}_m)+\tilde{G}_m(V_{rest}-\mathrm{V}_m)+I_{inj}\right]\\
\frac{dn}{dt} &= \alpha_n(\mathrm{V}_m)(1-\mathrm{n})-\beta_n(\mathrm{V}_m)\mathrm{n}\\
\frac{dm}{dt} &= \alpha_m(\mathrm{V}_m)(1-\mathrm{m})-\beta_m(\mathrm{V}_m)\mathrm{m}\\
\frac{dh}{dt} &= \alpha_h(\mathrm{V}_m)(1-\mathrm{h})-\beta_h(\mathrm{V}_m)\mathrm{h}
\end{aligned}$$

with the α and β kinetics given by:

$$\begin{aligned}
\alpha_n(V_m) &= \frac{(10 - V_m)}{100(e^{(10-V_m)/10} - 1)} \\
\beta_n(V_m) &= 0.125e^{-V_m/80} \\
\alpha_m(V_m) &= \frac{(25 - V_m)}{10(e^{(25-V_m)/10} - 1)} \\
\beta_m(V_m) &= 4e^{-V_m/18} \\
\alpha_h(V_m) &= 0.07e^{-V_m/20} \\
\beta_h(V_m) &= \frac{1}{e^{(30-V)/10} + 1}
\end{aligned}$$

Figures (5.1) and (5.2) defined this problem for use with the makecode.py scripts, with additional parameters in the α and β kinetics given by:

$$\begin{aligned}
\alpha_n(V_m) &= \frac{an_C(an_{V2} - V_m)}{(e^{an_{V3}(an_{V2}-V_m)} - 1)} \\
\beta_n(V_m) &= bn_C e^{-bn_{V1}V_m} \\
\alpha_m(V_m) &= \frac{am_C(am_{V1} - V_m)}{(e^{am_{V3}(am_{V1}-V_m)} - 1)} \\
\beta_m(V_m) &= bm_C e^{-bm_{V1}V_m} \\
\alpha_h(V_m) &= ah_C e^{-ah_{V1}V_m} \\
\beta_h(V_m) &= \frac{bh_C}{e^{bh_{V2}(bh_{V1}-V_m)} + 1}
\end{aligned}$$

This now becomes a dynamical parameter estimation problem with 4 dynamical variables and 22 unknown parameters.

6.1.1 Hodgkin Huxley Results

As seen in Table 6.1, and Figures 6.1 and 6.2, the dynamic state and parameter estimation technique accurately estimates the unmeasured parameters in this model, as well as the unmeasured gating variables, m, n and h. Importantly, the control term, u(t), is reduced to zero, and the R-value is 1 at every time point.

Table 6.1: HH Model: Data Parameters and Estimated Parameters.

Param	Data	T=500	T=5000	Param	Data	T=500	T=5000
C_m	1.0	1.00	1.00	bm_{V1}	0.056	0.0556	0.0557
g_{Na}	120.0	120.9	119.9	ah_C	0.07	0.0693	0.0701
E_{Na}	115.0	115.0	115.0	ah_{V1}	0.05	0.0498	0.050
g_K	36.0	36.0	36.0	bh_C	1.0	1.00	1.00
E_K	-12.0	-12.0	-12.0	bh_{V1}	30.0	29.99	30.0
g_M	0.3	0.30	0.30	bh_{V2}	0.1	0.100	0.100
E_{rest}	10.613	10.618	10.648	an_C	0.01	0.0099	.009997
am_{V1}	25.0	25.0	25.0	an_{V2}	10.0	9.98	10.0
am_{V3}	0.1	0.100	0.099	an_{V3}	0.1	0.100	0.0999
am_C	0.10	0.100	0.100	bn_C	0.125	0.125	0.125
bm_C	4.0	4.00	4.00	bn_{V1}	0.0125	0.01247	0.0125

6.1.2 Hodgkin-Huxley Discussion

This is an important result in comparison to the successful Morris-Lecar results for a few reasons. First, the Morris-Lecar problem was solved using the SNOPT optimization software, whereas Hodgkin-Huxley was solved using IPOPT optimization software. These are both widely available software packages for large-scaled nonlinear optimizations, but utilize different algorithms. The fact that dynamical state and parameter estimation problems can be solved using either software package gives considerable flexibility, and an easy way to check solutions.

Second, the Hodgkin-Huxley problem is more complicated than the Morris-Lecar problem, with more state variables and more unknown parameters. As neurobiological models become more complex, the ability of the dynamical state and parameter estimation method to solve larger systems may become difficult.

Third, and perhaps most importantly, the Hodgkin-Huxley model is widely used as the starting point for modeling neurons in experimental systems. Provided that this is in fact a good physical model of whichever experimental systems is being studied, these methods are expected to give an accurate estimation of unmeasured parameters and states.

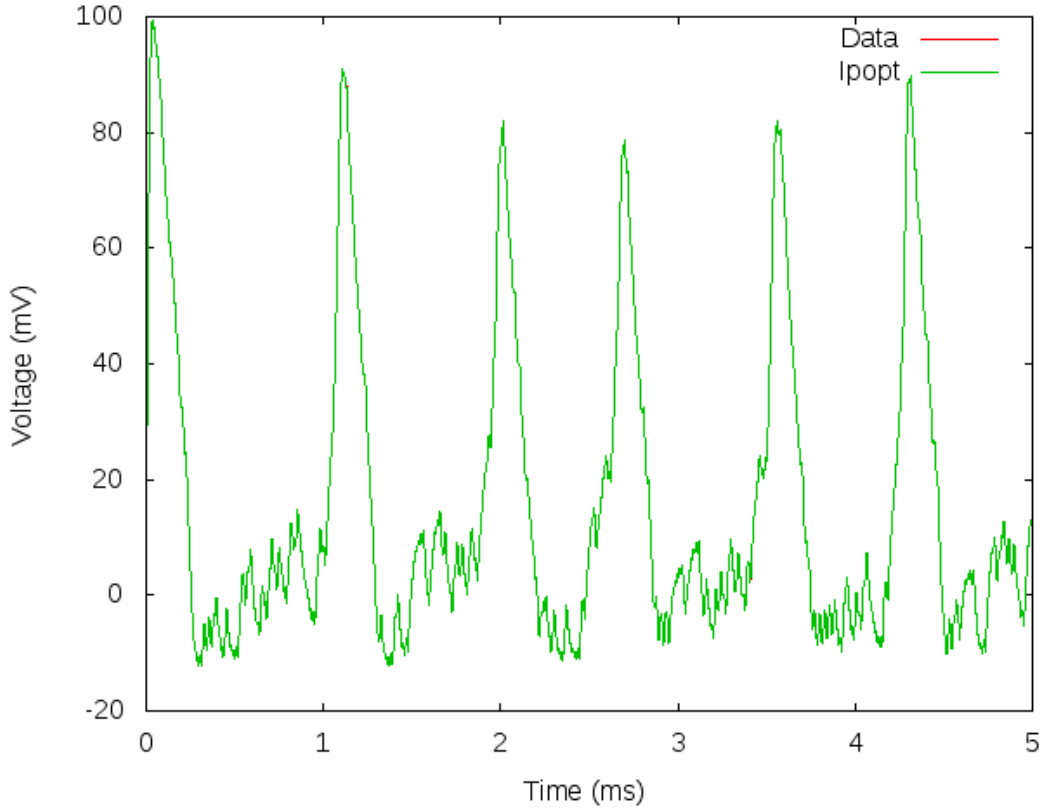

Figure 6.1: Hodgkin-Huxley dynamical parameter estimation of voltage. An injected current patterned off the chaotic signal from a Lorenz system was used in order to explore the dynamical range of the neuron. Note that the estimated voltage is indistinguishable from the 'data' trace, as expected.

6.2 LP Neuron Model

The California Spiny Lobster (Panulirus interruptus) is a decapod crustacean, shown in Figures (6.3) and (6.4), and is studied by neurobiologists because of the characteristics of two central pattern generators (CPGs) contained in its nervous system. These CPGs, the pyloric and gastric mill CPGs are part of the stomatogastric ganglion, Figure (6.5), and control muscles the produce chewing and muscles that dilate and constrict the stomach.[46]

The cellular and synaptic properties found in the stomatogastric ganglion are similar to those found in all nervous systems. This, and the fact that the

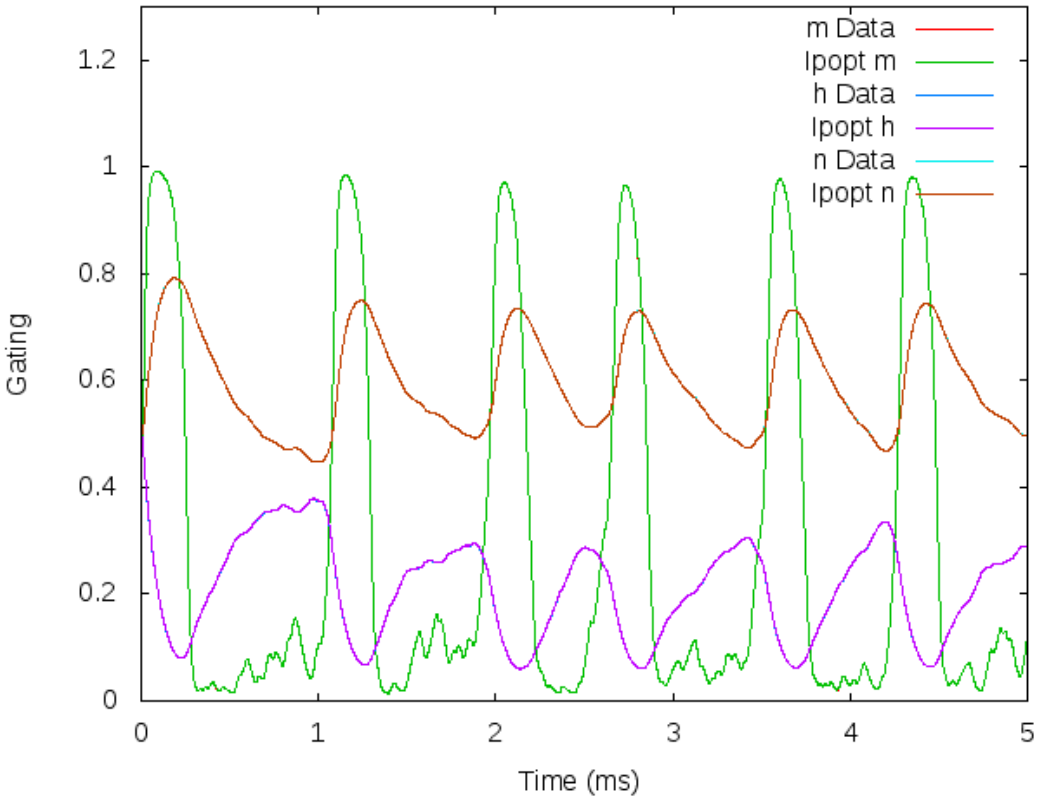

Figure 6.2: Hodgkin-Huxley dynamical parameter estimation of unobserved gating variables. This twin experiment estimation remarkably gives unobserved state variables that are indistinguishable from the actual values.

system is relatively small, make this system a well-studied neural circuit.

The pyloric central pattern generator is a small and autonomous network of fourteen neurons in the stomatogastric ganglion. It produces a rhythmic pattern controlling the pylorus, and underlies the production of most rhythmic motor patterns. One lateral pyloric (LP) neuron is present in the circuit, and is relatively easy to identify and record from. This neuron is a conditional burster, with burst durations and frequency consistent across preparations. Figures (6.7) and (6.8) show typical LP neuron behavior in response to injection of DC current. As the current increases (Figure (6.8)), the LP neuron exhibits bursting behavior, many spikes in succession with long recovery periods in between each burst. Reference [38] presents a model for the lateral pyloric (LP) neuron of the pyloric

Figure 6.3: California Spiny Lobster (Panulirus interruptus).

central pattern generator, Figure (6.6).

6.2.1 LP Neuron Model

The reference [38] model is a conductance based model in the Hodgkin-Huxley type vein, but with significantly more complexity to account for the bursting behavior. The ionic currents are grouped into two compartments, denoted by 'soma' and 'axon', in order to depict some spatial difference between where the currents interact with the cell membrane. The 'soma' compartment contains a transient calcium current and a slow calcium current (I_{Ca}), a calcium induced potassium current (I_{KCa}), a hyperpolarization-activated inward 'h' current (I_h) and a transient potassium-delayed spike initiation 'A' (I_A) current. The 'axon' compartment contains the familiar sodium (I_{Na}) and potassium (I_{Kd}) currents, as

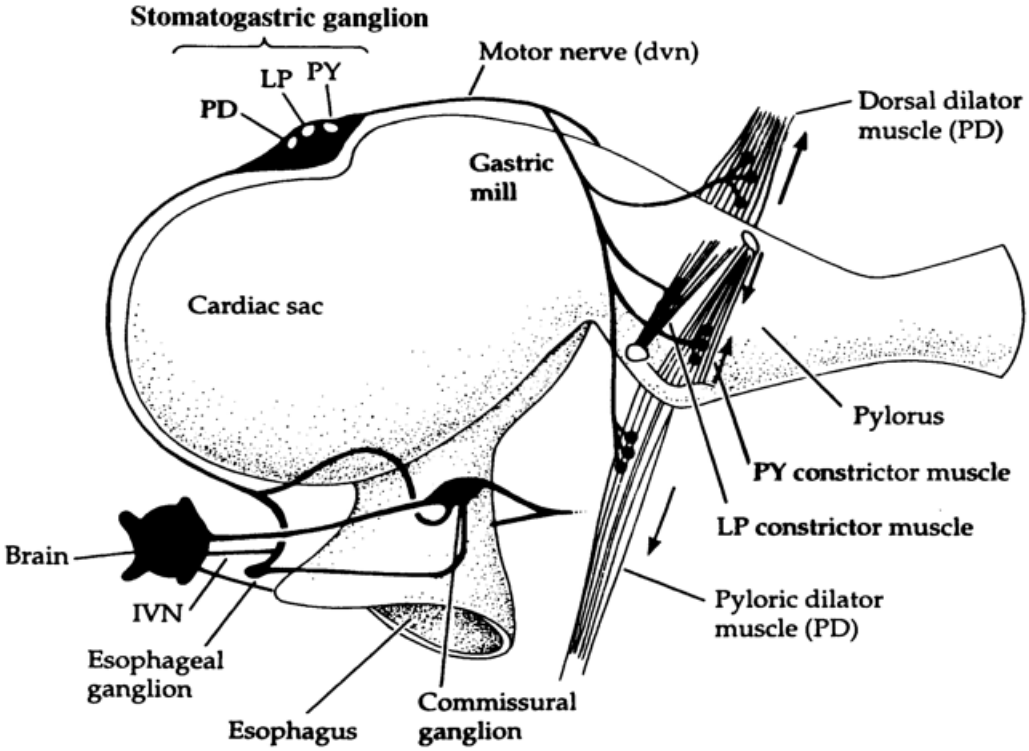

Figure 6.4: A cartoon of relevant California Spiny Lobster biological systems.

well as a spike rate adaptation current (I_M).

Both compartments include a leak current (I_{leak}) and are coupled together with an ohmic connection (I_{VV}). Current is injected into the 'soma' compartment to coincide with the experimental setup. With these currents, the voltage dynamics are:

$$\begin{aligned} \frac{dV_{axon}}{dt} &= \frac{1}{\mathrm{C}_a}\left[-I_{Na} - I_{Kd} - I_M - I_{leak,a} + I_{VV}\right] \\ \frac{dV_{Soma}}{dt} &= \frac{1}{\mathrm{C}_s}\left[-I_{Ca} - I_{KCa} - I_A - I_h - I_{leak,s} - I_{VV} + I_{scale}(I_{DC} + I_{syn})\right] \end{aligned}$$

The bursting characteristics of this neuron are modeled using calcium channels, with three calcium-dependent currents. The calcium concentration is described by a first order kinetic equation,

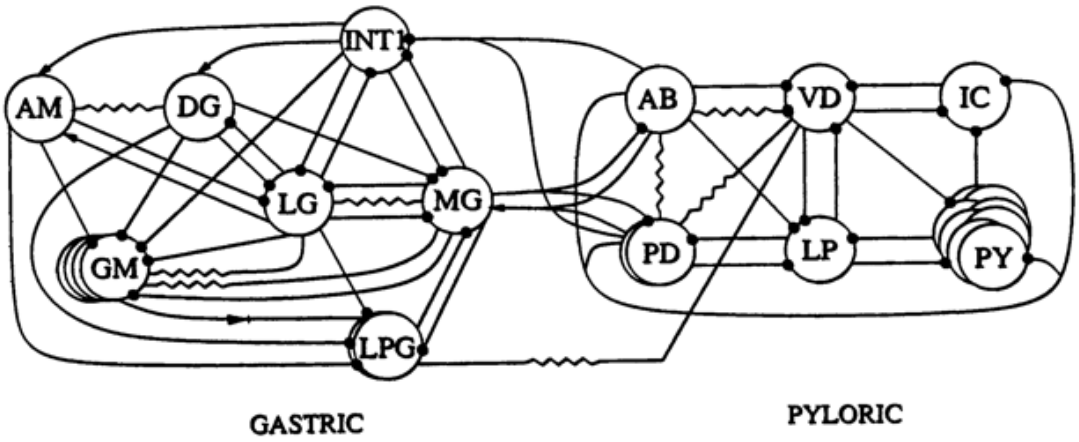

Figure 6.5: California Spiny Lobster stomatogastric ganglion system.

$$\frac{d[Ca]}{dt} = -c_{ICa} I_{Ca} - k_{Ca}([Ca] - [Ca]_0)$$

and the calcium current is given by the Goldman-Hodgkin-Katz equation,

$$I_{Ca} = (g_{CaT} m_{CaT} h_{CaT} + c_{CaS} m_{CaS}) \frac{[Ca] \exp[\frac{V_{Soma}}{RT/F}] - [Ca]_{out}}{\exp[\frac{V_{Soma}}{RT/F}] - 1.0} P_{Ca} V_{Soma}.$$

All ionic currents (except as noted) are given by:

$$I_x = g_x m^p h^q (V - V_x),$$

with the activation and inactivation variables governed by:

$$\frac{dm}{dt} = (a_m(1-m) - b_m m)$$
$$\frac{dh}{dt} = (a_h(1-h) - b_h h)$$

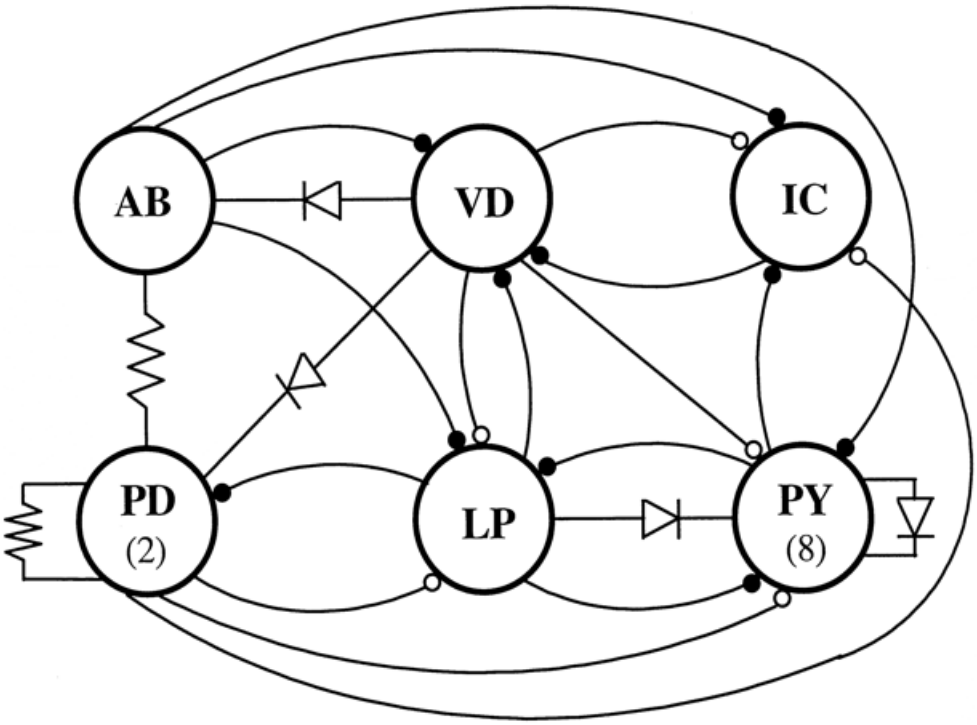

Figure 6.6: California Spiny Lobster pyloric central pattern generator.

for I_{Na} and I_{Kd} and

$$\begin{aligned} \frac{dm_x}{dt} &= (m_{\infty x}(V) - m_x)k_{mx} \\ \frac{dh_x}{dt} &= (h_{\infty x}(V) - h_x)k_{hx} \end{aligned}$$

for I_{Ca}, I_{KCa}, I_A, I_h, and I_M.

The specific kinetics for each of these currents is shown in Figure (6.11).

All told, the LP neuron model includes 16 dynamical equations and 73 parameters. A resulting numerical integration of this model, using parameter values given in reference [38], is shown in Figure (6.12).

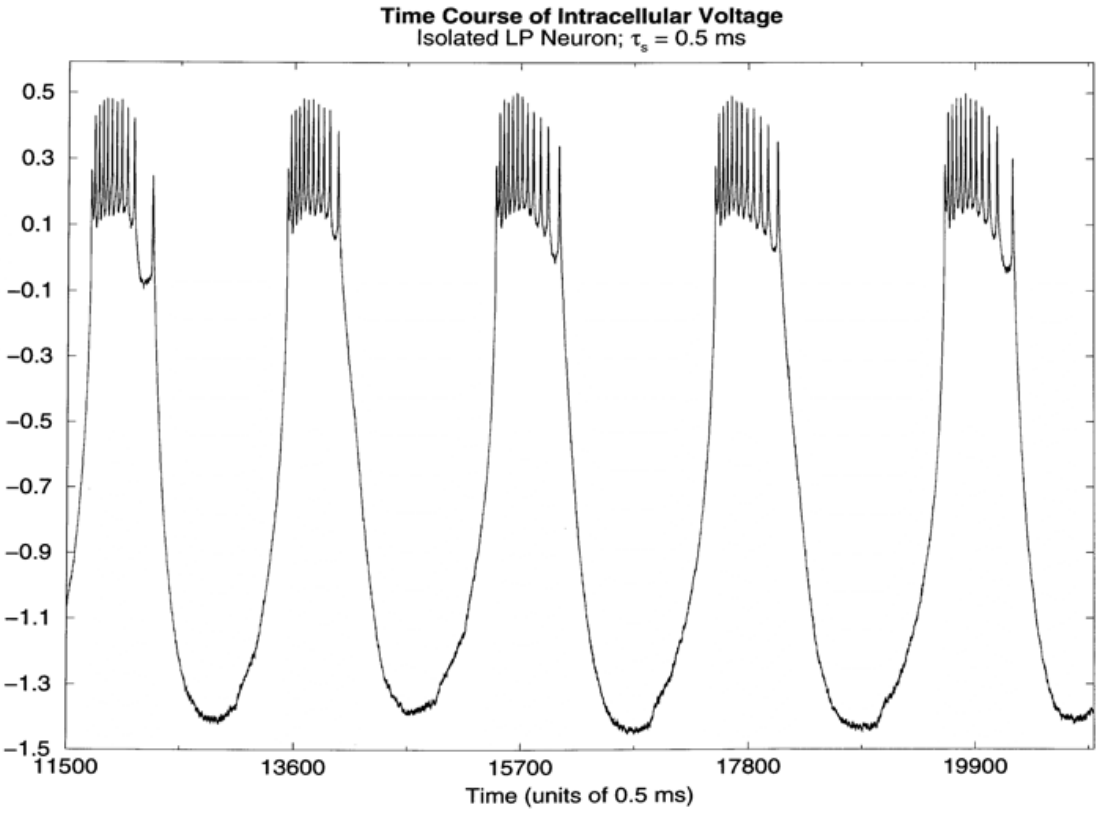

Figure 6.7: California Spiny Lobster pyloric central pattern generator LP neuron voltage trace.

6.2.2 LP Neuron Goals

The LP neuron model presents significant challenges, compared with the Hodgkin-Huxley model. The model is significantly more complicated in terms of number of equations and unknown parameters, but also due to the fact that the system is stiff, characterized by significantly different time scale dynamics for the membrane voltages compared to the calcium concentration dynamics. Also, the 73 parameters as written are unscaled, and include typical units for the quantity of interest (e.g., mV for all reversal potentials). As a result, the parameter set includes values that range over eight orders of magnitude, which may cause difficulty in the direct method linear algebra methods employed by the optimization software. In addition, reference [38] discussed two other challenges with this model:

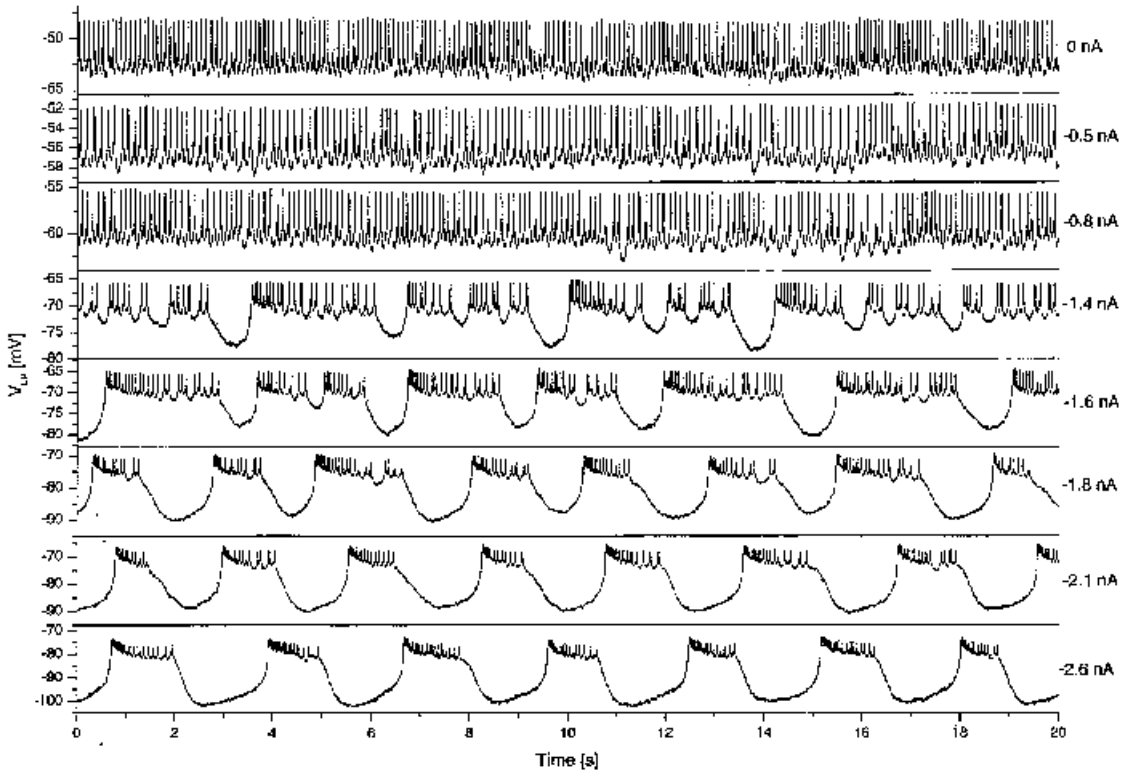

Figure 6.8: California Spiny Lobster pyloric central pattern generator LP neuron voltage traces for various input currents.

- Sensitivity of this model system to small changes in the parameter set, with many parameter sets potentially giving similar model results, specifically with the parameters presented in Figure (6.10)

- Chaotic behavior connected to the calcium dynamics, as first suggested in [17], to explain the range of behavior observed for changing input currents. Reference [38] calculates the Lyapunov exponents in this model and confirms the existence of at least one positive exponent, explaining the chaotic nature of the dynamics.

With these challenges in mind, the goal of a complete parameter and state estimation based solely on a voltage measurement is too aggressive. For instance, coupling the soma voltage data back into the model may not be enough to reduce the positive conditional Lyapunov exponents to non-positive values, which

I_{CaT}	V_{mCaT}	s_{mCaT}	V_{hCaT}	S_{hCaT}	k_{mCaT}	k_{hCaT}	V_{khCaT}	s_{khCaT}
	15 mV	−9.8 mV	−40 mV	3.2 mV	45 Hz	20 Hz	−15 mV	−10 mV
I_{CaS}	V_{mCaS}	s_{mCaS}	k_{mCaS}					
	29.13 mV	−4.431 mV	60.86 Hz					
I_{KCa}	c_{mKCa}	V_{mKCa1}	s_{mKCa1}	V_{mKCa2}	s_{mKCa2}	c_{hKCa1}	c_{hKCa2}	$kmKCa$
	2.5 µM	0 mV	−23 mV	−16 mV	−5 mV	0.7 µM	0.6 µM	600 Hz
	k_{hKCa}	f						
	35 Hz	0.6 mV/µM						
I_A	V_{mA}	s_{mA}	V_{hA}	s_{hA}	k_{mA}	k_{hA1}	k_{hA2}	V_{kha2}
	−12 mV	−26 mV	−62 mV	6 mV	140 Hz	50 Hz	3.6 Hz	−40 mV
	s_{khA2}	V_{aA}	s_{aA}					
	−12 mV	7 mV	−15 mV					
I_h	V_{mh}	s_{mh}	V_{kmh}	s_{kmh}	k_{mh}			
	−70 mV	8 mV	−110 mV	−21.6 mV	0.33 Hz			
V_x	V_{Na}	V_{Kd}	V_{KCa}	V_A	V_h	V_M	V_{leak}	
	50 mV	−72 mV	−72 mV	−72 mV	−20 mV	−80 mV	−50 mV	
other	RT/F	g_{Na}	g_{Kd}	P_{Ca}	$[Ca]_{out}$	$[Ca]_0$		
	11.49 mV	715 µS	143 µS	1.56 1/mM	15120 µM	0.02 µM		

Figure 6.9: LP neuron parameters used in numerical integration.

Conductances	g_{CaT}	g_{CaS}	g_{KCa}	g_A	g_h	$g_{leak,a}$	$g_{leak,s}$
	11.23 µS	6.428 µS	149.7 µS	72.08 µS	1.142 µS	0.1004 µS	0.06211 µS
Other	C_a	C_{ICa}	k_{Ca}	V_{shift}	g_w	C_s	I_{scale}
	11.48 nF	505 M/As	17.3 Hz	−9.343 mV	0.4017 µS	5.439 nF	1.223
M current	g_M	V_{mM}	s_{mM}	k_{mM}	V_{kmM}	s_{kmM}	
	26.13 µS	−26.99 mV	−5.957 mV	0.1387 Hz	−60.58 mV	−13.26 mV	

Figure 6.10: More LP neuron parameters used in numerical integration.

is necessary for synchronization of the data to the model [4]. Without this synchronization, the cost function of the optimization will not be smooth, making detection of a global minimum very difficult, if not impossible.

Instead, the goal here is to explore the strengths and weaknesses of the dynamical state and parameter estimation method, using the full LP neuron model as a template. This work will include a comparison of optimizations based on coupling one state variable, two state variables, and all sixteen state variables of data to the model, using the dynamical state and parameter estimation method, as well as performing the minimizations using the unconstrained minAzero path integral method. Also, I will explore reduction of the number of 'free' parameters in the model to determine how much of a difference this makes.

	p	q	a_m	b_m	a_h	b_h
I_N	3	1	$0.32\frac{V_{axon}+52}{1-\exp\left(-\frac{V_{axon}+52}{4}\right)}$	$0.28\frac{V_{axon}+25}{\exp\left(\frac{V_{axon}+25}{5}\right)-1}$	$0.128\exp\left(-\frac{V_{axon}+48}{18}\right)$	$\frac{4}{\exp\left(-\frac{V_{axon}+25}{5}\right)+1}$
I_{Kd}	4	0	$0.32\frac{V_{axon}+50}{1-\exp\left(-\frac{V_{axon}+50}{5}\right)}$	$0.5\exp\left(-\frac{V_{axon}+55}{40}\right)$		
	p	q	m_∞	h_∞	k_m	k_h
I_{CaT}	1	1	$\frac{1}{1+\exp\left(\frac{V_{soma}-V_{mCaT}}{s_{mCaT}}\right)}$	$\frac{1}{1+\exp\left(\frac{V_{soma}-V_{hCaT}}{s_{hCaT}}\right)}$	k_{mCaT}	$\frac{k_{hCaT}}{1+\exp\left(\frac{V_{soma}-V_{khCaT}}{s_{khCaT}}\right)}$
I_{CaS}	1	0	$\frac{1}{1+\exp\left(\frac{V_{soma}-V_{mCaS}}{s_{mCaS}}\right)}$		k_{mCaS}	
I_{KCa}	1	1	$\frac{[Ca]}{c_{mKCa}+[Ca]}$ $\times\frac{1}{1+\exp\left(\frac{V_{soma}-(V_{mKCa1}-f^*[Ca])}{s_{mKCa1}}\right)}$ $\times\frac{1}{1+\exp\left(\frac{V_{soma}-(V_{mKCa2}-f^*[Ca])}{s_{mKCa2}}\right)}$	$\frac{c_{hKCa1}}{c_{hKCa2}+[Ca]}$	k_{mKCa}	k_{hKCa}
I_A	3	1,1	$\frac{1}{1+\exp\left(\frac{V_{soma}-V_{mA}}{s_{mA}}\right)}$	h_{A1} & h_{A2}: $\frac{1}{1+\exp\left(\frac{V_{soma}-V_{hA}}{s_{hA}}\right)}$	k_{mA}	$h_{A1}: k_{hA1}$ $h_{A2}: \frac{k_{hA2}}{1+\exp\left(\frac{V_{soma}-V_{khA2}}{s_{khA2}}\right)}$
I_h	1	0	$\frac{1}{1+\exp\left(\frac{V_{soma}-V_{mh}}{s_{mh}}\right)}$		k_{mh} $\times\left(1+\exp\left(\frac{V_{soma}-V_{kmh}}{s_{kmh}}\right)\right)$	
I_M	1	0	$\frac{1}{1+\exp\left(\frac{V_{axon}-V_{mM}}{s_{mM}}\right)}$		$\frac{k_{mM}}{1+\exp\left(\frac{V_{axon}-V_{kmM}}{s_{kmM}}\right)}$	

Figure 6.11: Activation and inactivation functions for various LP neuron model currents.

6.2.3 LP Neuron Results

These computational 'twin' experiments use numerically integrated data sets. All integrations were performed using the 'odeint' function with Python's Scipy module and checked with the 'NDSolve' function in Wolfram Research's Mathematica program. This check is necessary to ensure that the integration time step is small enough to account for the problem stiffness due to the different dynamic time scales for the voltage and calcium dynamics.

Naive DSPE coupling

With simple models, the portion of data that is sampled for DSPE is not very important. The data generally must be long enough to include an interesting phase space portrait of the system, but for smaller dimension models, this is not hard. For example, in the Hodgkin-Huxley example, a data set that include 2-3 voltage spikes completely explores the dynamic range of all state variables.

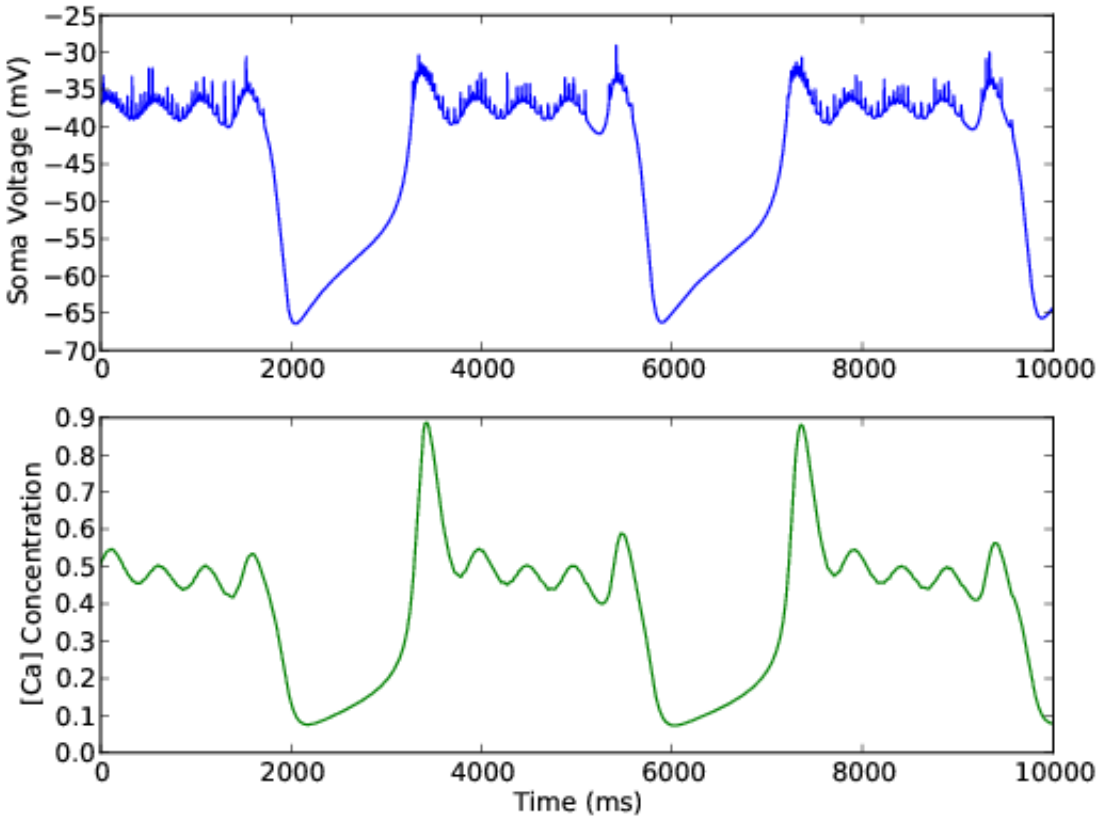

Figure 6.12: LP neuron voltage and calcium concentrations for typical DC current. The voltage shows the same general characteristics as real data shown in Figures (6.7) and (6.8).

This approach does not work for the LP neuron, due to the system stiffness. The interesting dynamics associated with the calcium concentration generally occurs when the neuron is not spiking, and vice versa. As a result, if a portion of data that only includes spiking, but no calcium concentration transient, is used, very little information is available to describe the full dynamics of the model, thus producing a bad fit.

As an illustration of this difficulty, a portion of integrated data which includes five voltage spikes is used in the normal DSPE procedure. This is done with unknown parameters from each of figures (6.9) and (6.10), and for only the soma

voltage coupling as well as a soma/calcium concentration combination.

As expected, the coupled variables give good matches to the data; in figure (6.13), the soma voltage and calcium concentrations both match up as a result of the DSPE procedure. However, figure (6.14) shows that this is not a good fit, as the R-value is not equivalently one throughout the time sequence. This is reflected in tables (6.2) and (6.3), which show generally poor parameter fitting.

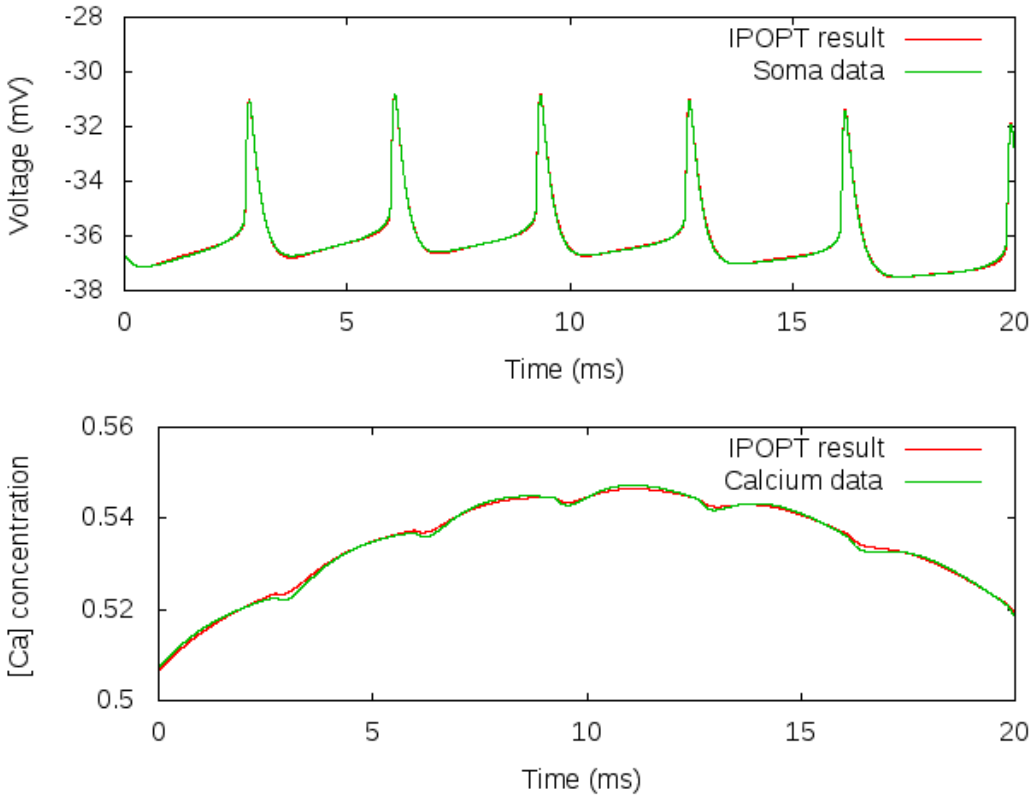

Figure 6.13: Results for DSPE of lobster LP neuron. Although both the coupled variable, V_{soma} and [Ca] produce good fits to the data, the estimated parameters and unobserved states are not identical to the data.

minAzero V_{soma} and [Ca]

As an alternative to the DSPE method, I experimented as well with the minAzero code that I developed. For the same data set used previously, with 5 spikes and no calcium transient, this code gives generally similar results: the

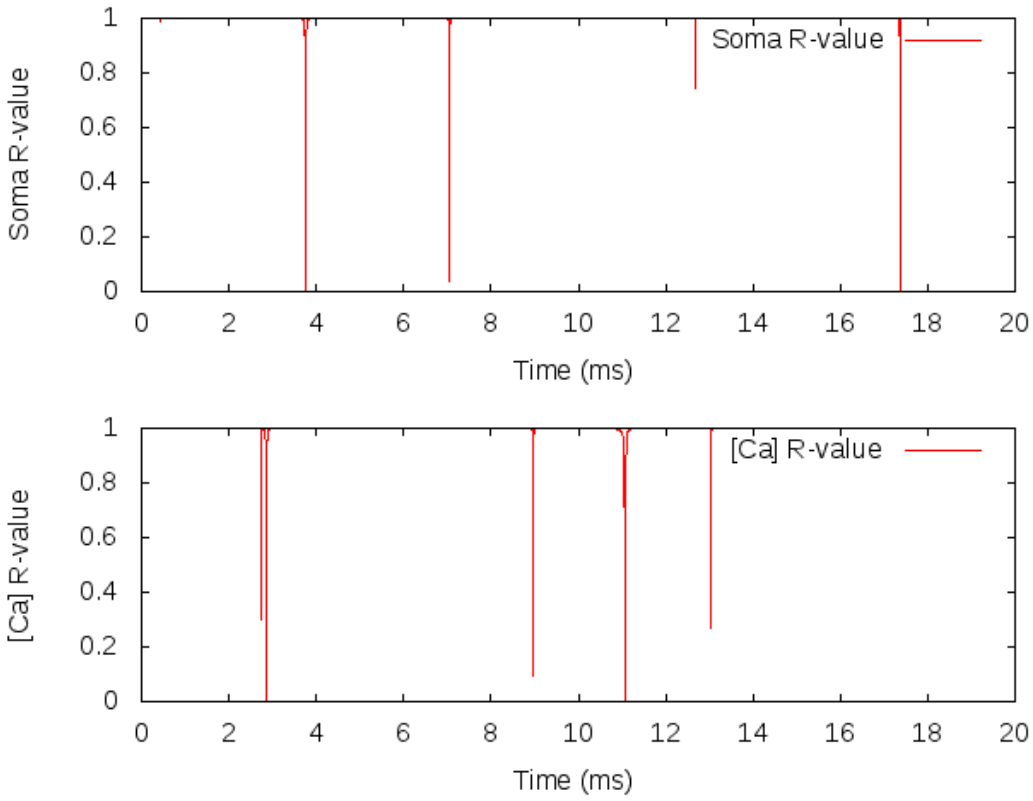

Figure 6.14: R-value results for DSPE of lobster LP neuron. The coupling overwhelms the model at a few points, showing that this is a bad fit.

measured variables are matched well, but the unmeasured variables and parameters are not. This supports the reference [38] conclusion that this model system has many parameter sets that potentially give similar model results. One way to confirm this is to utilize a data set that explores more of the dynamic range of the system.

DSPE: longer data set

In order to explore a larger portion of the dynamic range of the model, data must be used that includes activation of the calcium currents. For instance, in Figure (6.12), the range from 1800-3200 ms is a calcium transient characterized by large changes in the calcium concentration and no voltage spiking. In this region, the model state variables associated with the three calcium currents are

Table 6.2: LP Neuron Model: Data parameters and estimated parameters for coupling Soma voltage and Soma/Calcium using dynamic state and parameter estimation.

Param	Data	V	V, [Ca]	Param	Data	V	V, [Ca]
g_{Kca} μ**S**	166.4	500.0	500.0	$g_{leak,s}$ μ**S**	0.069	0.10	0.10
g_A μ**S**	80.1	250.0	250.0	g_M μ**S**	26.1	10.3	35.0
g_h μ**S**	12.0	0.67	0.66	k_M Hz	0.1387	0.10	0.10
$g_{leak,a}$ μ**S**	0.10	0.13	0.18	V_M mV	-26.99	-52.3	-13.1
C_{ICa} **M**/**A**s	504	1489	225.9	s_M mV^{-1}	-0.168	-5.0	-5.0
k_{Ca} Hz	17.0	1.0	10.6	V_{kM} mV	-60.6	-63.5	-76.0
g_{VV} μ**S**	0.40	0.42	0.43	s_{kM} mV^{-1}	-0.075	-5.00	-2.65

activated, and also undergo transients.

A data set including this transient is necessary in order to include the full dynamics of the model. For the same reason, a portion of the data set that includes voltage spiking must also be used. As a result, in order to explore the full dynamics of the LP neuron model, a data set encompassing approximately 2000 ms of data needs to be used.

A limitation of the DSPE method is that a good parameter fit requires adequate sampling of the data. This means that the integration time step must capture the fastest dynamics of the system - in this case the voltage spiking. Experience with the Morris-Lecar and Hodgkin-Huxley models has shown that an integration time step of .01 ms is appropriate to capture these dynamics; use of a .01 ms time step here requires 200,000 points of data in the optimization.

Unfortunately, this is too many points for many current hardware/software implementations to handle. In the given configuration, with T data points, this model has 19T+33 variables and 16T constraints, with 148T non-zero Jacobian entries and 58T+72 non-zero Hessian entries. All of these Jacobian and Hessian non-zero entries must be calculated at each optimization iteration, each constraint must be checked, and each variable updated. With 200,000 data points, this program becomes a memory-hog and will not run without some form of high

performance computing parallelization.

6.2.4 LP Neuron Remarks

Revisiting the goals of this section, full matching of all states and parameters in the LP neuron model is an aggressive goal, and has not yet been met. However, many valuable lessons about DSPE have been learned through the use of this complicated model.

Variable Scaling

First, I determined the necessity of proper variable scaling within the model. The optimization software works in part by performing linear algebra operations on large matrices; if the problem is not well scaled, the software may make some inappropriate approximations during these operations. For instance, the state variables for voltage typically range in the 10s of mV range, [Ca] in 1-5 $\mu\mathbf{M}$ range, and voltage gating variables between zero and one. These units are sufficient without scaling, as they only range over two orders of magnitude. However, some of the calcium current dynamical variables range over a much smaller scale, the smallest being order 10^{-4} at its peak. Six orders of magnitude can easily become a problem, as evidenced by warning and error messages from the optimization software.

This scaling issue is combatted with explicit rescaling of the variables within the 'equations.txt' file, and corresponding adjustments to the scale of related parameters. If the product of a state variable with corresponding parameters is too small, then it probably does not contribute to the observed dynamics, and can be deleted. In fact, I removed one of the calcium currents in the reference [38] LP neuron model which did not noticeably contribute to either the calcium or membrane voltage dynamics.

Exponential Scaling

Scaling issues exist even with proper variable scaling, due to the functional make-up of the LP neuron model. The model kinetics, typical of many neurobi-

ological conductance models, includes many exponential and sigmoidal terms, for example, terms of the form $x/(1 \pm e^x)$. These terms pose two different issues, depending on the sign in the denominator.

For the minus case, $x/(1 - e^x)$ is a smooth function defined for all x. At x=0, this functional form gives 0/0, and simple calculus is need via L'Hôpital's rule to compute the functional value of -1. For functions of this kind that are included in the optimization, the C^{++} program will have difficulty calculating the function at x=0.

For the plus case, $x/(1 + e^x)$, the plus sign in the denominator ensures that division by zero is not possible, so the same problem does not apply. Here, the problem is in how SymPy performs derivatives and displays them in the C^{++} files. These derivatives are performed correctly, but not simplified, so that complicated expressions may be present in both the numerator and denominator. If the argument over the exponential, x, is a free variable (or a combination of free variables) in the optimization, then these exponential terms can become very large relatively quickly. Whereas the ratio of numerator to denominator may in fact be rather small, double precision accuracy may not be sufficient to accurately reflect the 'true' values of these functions.

Both of these problems may cause the optimization program to give error messages during program execution. To combat these errors, I built the ability to include functions into the Python scripts. These functions are input directly into equations.txt. For instance, the minus case above is the function called efunc:

$$efunc(a, b, c, d) == \frac{a(b + d)}{e^{c(b+d)} - 1}$$

This function is defined in a C^{++} file called myfunctions.cpp, along with all first and second derivatives with respect to the variables (a,b,c,d). This allows for coding within the C^{++} file to explicitly change the functional form close to any points that would give 0/0 (i.e., b+d=0). In this case, I used a simple three term Taylor expansion of the original function.

Any new function can be defined in this way and, if properly designated within 'equations.txt', used within a DSPE optimization. The implementation

of this functionality is a bit messy - the Sympy module handles the function as a string, and takes derivatives with respect to the function, but the Python code only handles the symbolic form of the optimization problem - the numerical work is done in the generated files, which call the functions in myfunctions.cpp. Therefore, the Python code includes regular expressions to massage these functions into the correct form for output to the generated files.

6.2.5 LP Neuron Conclusions

The ultimate goal of DSPE is to take measurements from an experimental system and estimate unmeasured states and parameters. A necessary first step in this process is to construct a model that accurately describes the physics of the experimental system, and use this model to conduct twin experiments with numerically generated data. In the case of the LP neuron, these twin experiments have given some valuable insight into the strengths and weaknesses of the DSPE method, but ultimately show that experiments with real data are not yet feasible.

A number of approaches are possible to continue this work. Since the model appears to be overdetermined (i.e., multiple parameter sets produce similar dynamics), the model must be examined in detail to reduce the degrees of freedom. This is non-trivial for such a complicated model, but is necessary in order for this, or any other method, to work.

In addition, the large amount of data needed to perform this method must be addressed. One solution is to use an adaptive time step routine, whereby data sampling rates can be slower during a 'slow' transient, such as the calcium transient, and faster for the 'fast' dynamics, i.e. voltage. This allows for sampling over the entire dynamic range of all state variables, but requires significantly fewer data points.

6.3 Three Neuron Network

Single neuron models such as Morris-Lecar, Hodgkin-Huxley, and the LP neuron model are interesting for determining the dynamic response and biophysical

properties of neurons, but more interesting questions involve multiple neurons and how they are connected in networks. For reference, the human brain has roughly 10^{10} neurons which form 10^{15} connections - modeling this is a distant goal; instead I will start with a simple three-neuron circuit.

6.3.1 Network Model

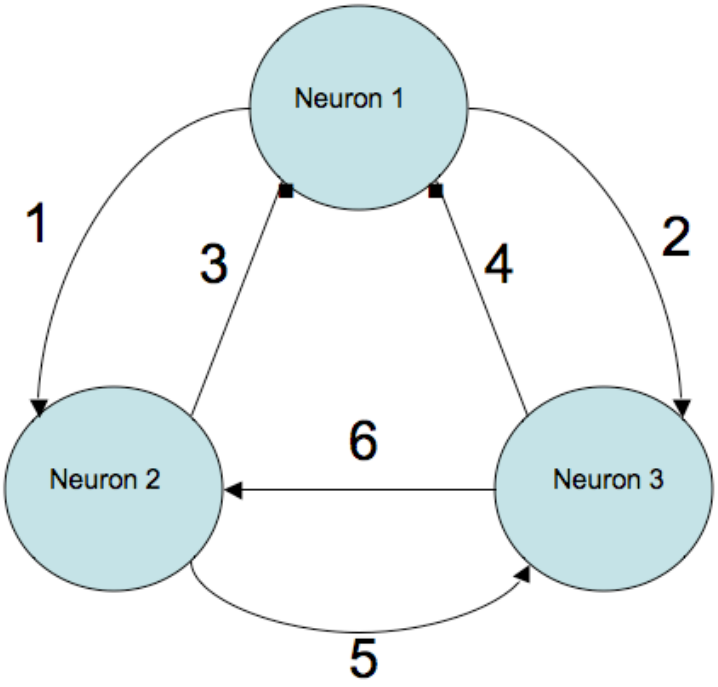

Figure 6.15: Schematic of three neuron network.

Each neuron is connected to each other neuron with two connections, as shown in figure (6.15). Inhibitory synapses (i.e, these lower the membrane voltage and thus make a neuron less likely to spike) are shown with a dot; excitatory synapses are shown with arrow. Synapses are described by a standard model,

$$I_{syn} = g_{syn} S (V_{rev} - V_{post}) \tag{6.1}$$

$$\frac{dS}{dt} = \frac{1}{\tau} \frac{0.5(1 + tanh((V - V_{sa})/V_{sb})) - S}{s_c - 0.5(1 + tanh((V - V_{sa})/V_{sb}))} \tag{6.2}$$

and each individual neuron is described by standard Hodgkin-Huxley dynamics with known parameters, so that the only unknown parameters are associated with the synapses. All told, this model includes 18 state variables (3 membrane voltages, 3x3 gating kinetics, 6 synapses) and 36 parameters.

6.3.2 Network Goals

In a real experimental network, measurement of membrane voltages from multiple neurons is typically not possible, but with twin experiments where all the 'data' is known, it is. To try to match up with reality in some form, the initial goal is to determine how much information about the network synapses can be determined by only one membrane voltage measurement. But since more than one measurement is possible in this twin experiment, all permutations of two membrane voltage measurements will also be used in the dynamical parameter estimation procedure.

6.3.3 Network Results

Six experiments were performed - three single voltage and three dual voltage, and the results are consistent, which should be expected for an almost symmetrical system. In all experiments, synaptic dynamics for synapses transmitting information *to* a neuron with measured membrane voltages are fit exactly, along with good parameter fits. However, all other synapses and all unmeasured membrane voltages, along with associated parameters, do not have good fits.

An example is shown in table (6.5). Here, the neuron membrane voltage is measured for neuron numbers one and three, but not neuron number two. The parameters corresponding to the synaptic currents, equation (6.1) in the neuron number two voltage equation are labeled by indices one and six; as seen in the table, all parameters are fitted exactly for synaptic indices two through five, but quite terribly for synapses one and six. These results are also true for the parameters in equation (6.2), table (6.6), with the exception of the V_{sa} terms, which is surprising.

6.3.4 Network Conclusions

If the synaptic connections are thought of as pipes bringing information to and taking information away from neurons, then these results make sense. By knowing the exact voltage trace for a given neuron, the DSPE procedure can determine the synaptic parameters necessary to produce that voltage, but cannot determine exactly the unmeasured voltage. For synapses taking information away from the measured neuron, there is no way to know what the exact dynamics of the synapses are, without accurate knowledge of the membrane voltage of the neuron that the information is flowing to.

Table 6.3: LP Neuron Model: Data parameters and estimated parameters for coupling Soma voltage and Soma/Calcium using dynamic state and parameter estimation using alternate parameter set.

Param	Data	10^4
g_{Na} μ**S**	715	234
V_{Na} mV	50	65.2
V_K mV	-72	-56.3
g_{kd} μ**S**	143	38.4
V_h mV	-20	-5
V_{leak} mV	-50	-5
RT/F mV^{-1}	0.088	0.083
k_{mCaT} Hz	45	43.5
k_{hCaT} Hz	20	11.4
V_{mCaT} mV	15	24.1
V_{hCaT} mV	-40	-72.5
s_{mCaT} mV^{-1}	-0.102	-0.106
s_{hCaT} mV^{-1}	0.3125	0.539
$[Ca]_{out}$ μ**M**	15123	20000
V_{khCaT} mV	-15	-32.5
s_{khCaT} mV^{-1}	-0.1	-5.0
k_{mKCa} Hz	600	589
k_{hKCa} Hz	35	14.1
V_{mKCa1} mV	0	1.4
V_{hKCa1} mV	16	1
s_{mKCa1} mV^{-1}	-0.043	0.8
s_{hKCa1} mV^{-1}	-0.2	-0.23

Param	Data	10^4
f mV/μ**M**	0.6	5.0
c_{mKCa} μ**M**	2.5	0.11
c_{hKCa1} μ**M**	0.7	1.0
c_{hKCa2} μ**M**	0.6	0.1
$[Ca]_0$ μ**M**	0.02	0.005
k_{mA} Hz	140	50.0
k_{hA1} Hz	50	53.5
k_{hA2} Hz	3.6	5.4
V_{mA} mV	-12	-35.0
V_{hA} mV	-62	-38.2
V_{kha2} mV	-40	-77.1
V_{aA} mV	7	1.0
s_{mA} mV^{-1}	-0.04	-0.43
s_{hA} mV^{-1}	0.167	1.36
s_{kha2} mV^{-1}	-0.083	0.62
s_{aA} mV^{-1}	-0.067	0.07
k_{mh} Hz	0.33	0.01
V_{mh} mV	-70	55.0
V_{kmh} mV	-110	-128.2
s_{mh} mV^{-1}	0.125	2.53
s_{kmh} mV^{-1}	-0.046	-0.223
V_M mV	-80	-76.7

Table 6.4: LP Neuron Model: Data parameters and estimated parameters for coupling Soma voltage and Soma/Calcium using MinAzero code.

Param	Data	V	V, [Ca]	Param	Data	V	V, [Ca]
g_{Kca} μ**S**	166.4	500.0	500.0	$g_{leak,s}$ μ**S**	0.069	0.055	0.089
g_A μ**S**	80.1	250.0	250.0	g_M μ**S**	26.1	10.0	10.0
g_h μ**S**	12.0	0.94	0.85	k_M Hz	0.1387	0.10	0.10
$g_{leak,a}$ μ**S**	0.10	0.10	0.10	V_M mV	-26.99	-55.2	-54.8
C_{ICa} **M**/**A**s	504	829	230.6	s_M mV^{-1}	-0.168	-1.67	-1.39
k_{Ca} Hz	17.0	1.0	11.4	V_{kM} mV	-60.6	-73.4	-73.4
g_{VV} μ**S**	0.40	0.412	0.413	s_{kM} mV^{-1}	-0.075	-2.56	-2.56

Table 6.5: Three neuron network Model: Data parameters and estimated parameters for coupling neurons one and three. Information travels to the unmeasured neuron two, corresponding to conductance and potentials one and six. Results are similar for the other 24 parameters.

Conductances	Data	DSPE	Reversal Potentials	Data	DSPE
g_{s1}	0.4	2.05	V_{rev1}	-5	-51.1
g_{s2}	0.5	0.50	V_{rev2}	-5	5.04
g_{s3}	1.2	1.20	V_{rev3}	-70	-70.0
g_{s4}	0.6	0.60	V_{rev4}	-5	-4.97
g_{s5}	1.2	1.21	V_{rev5}	-70	-70.0
g_{s6}	0.5	1.28	V_{rev6}	-5	-92.1

Table 6.6: Three neuron network Model: Data parameters and estimated parameters for coupling neurons one and three. Information travels to the unmeasured neuron two, corresponding to conductance and potentials one and six.

Conductances	Data	DSPE
V_{sa1}	3	27.4
V_{sa2}	3	3.0
V_{sa3}	-0.03	-3.2
V_{sa4}	3	0.60
V_{sa5}	0.0175	-0.378
V_{sa6}	3	0.024
V_{sb1}	5	1.16
V_{sb2}	5	5.0
V_{sb3}	8	7.75
V_{sb4}	5	5.0
V_{sb5}	8	8.40
V_{sb6}	5	18.18

Reversal Potentials	Data	DSPE
τ_1	1	0.25
τ_2	1	1.0
τ_3	0.33	0.334
τ_4	1	1.0
τ_5	0.33	0.335
τ_6	1	0.42
s_c1	1.5	1.06
s_c2	1.5	1.50
s_c3	1.66	1.67
s_c4	1.5	1.50
s_c5	1.66	1.68
s_c6	1.5	1.00

Chapter 7

Advanced Parameter Estimation

The dynamical parameter estimation technique has proven to work well for simple non-linear systems. For systems with one or more positive Lyapunov exponents (i.e., systems that can exhibit chaotic behavior), more advanced numerical techniques are necessary to generate a good parameter estimation. In this chapter, I will discuss some advance techniques for dynamical parameter and state estimation that may be necessary for complicated problems. Specifically, I will discuss:

- The difficulties with dealing with large data sets
- How to fabricate a good starting guess for the optimization

7.1 Large Problem Size

The magnitude of the largest Lyapunov exponent puts an upper limit on the length of data that can be used. As an example, consider the Lorenz 1996 model [31, 32]:

$$\dot{x_j} = (x_{j+1} - x_{j-2}) * x_{j-1} - x_j + F$$

7.1.1 Example

For a time step of 0.01 and 0.1, various data sets are generated using different integration time steps. For time step of 0.01, the integrations diverge at point 2850, or 28.5 units of time. This is related to the Lyapanov exponents of the system, as it is independent of integration time step. This is a limit of this technique for chaotic systems: numerical integration techniques have error, and will diverge. In this case, the longest data set that can be used is for 28.5 time units.

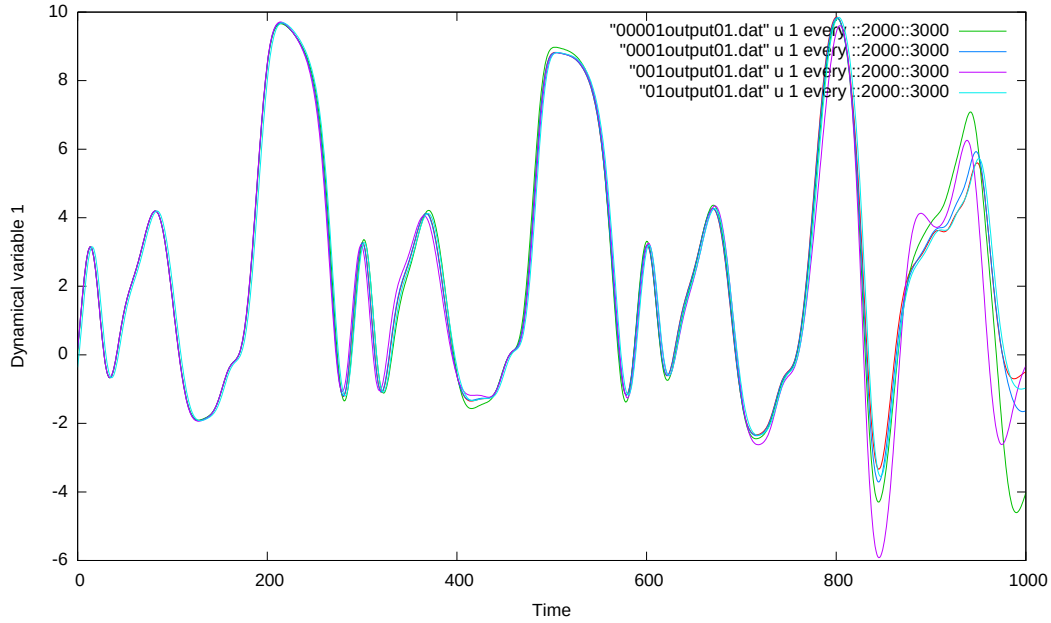

Figure 7.1: Integration results of the first variable of the Lorenz 1996 model, showing time 20 through 30. Note that regardless of the integration time step used, the integration diverges at 28.5 time units.

7.1.2 Discussion

This data set size limitation can be overcome by relaxing the tolerance criteria on the equality constraints of the optimization. In dynamical parameter estimation, equality constraints are given by the Hermite-Simpson integration map for the system: each time step is related to the previous time step by the integration

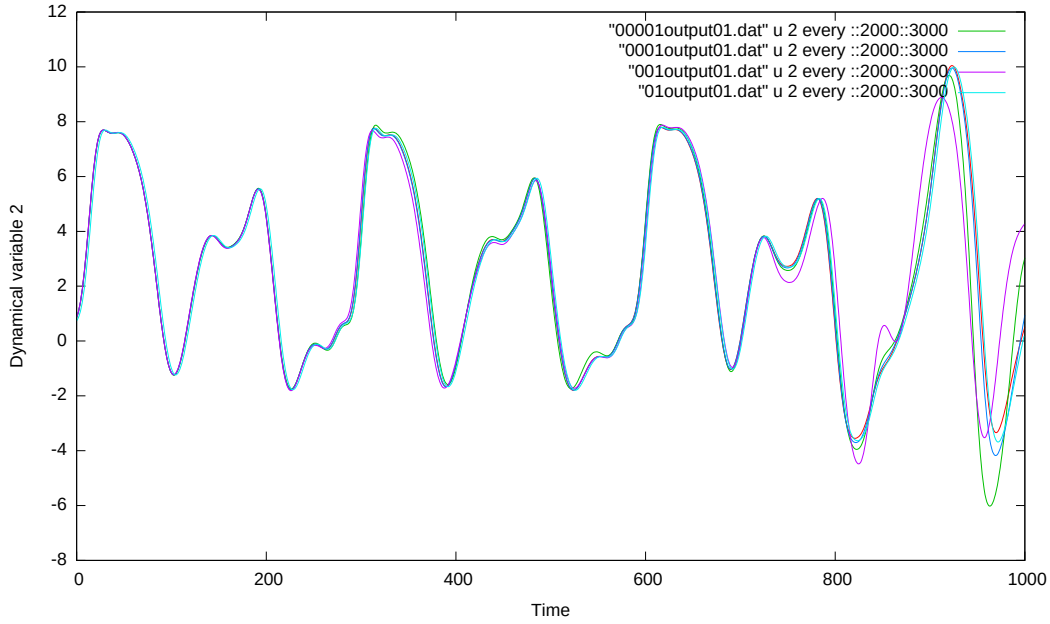

Figure 7.2: Integration results of the second variable of the Lorenz 1996 model, showing time 20 through 30. Note that regardless of the integration time step used, the integration diverges at 28.5 time units.

rule. For a strict equality constraint to be valid, the Hermite-Simpson rule must give identical results to the integration technique used to generate the data for a twin experiment. Since for chaotic systems, a given integration technique will diverge for different integration time steps, requiring a specific integration rule (Hermite-Simpson) to match whichever integration technique is used to generate the data would be impossible. This argument is similarly valid for data generated from a real system, where no numerical integration scheme was used! To mitigate this, the Hermite-Simpson rule used within the optimization can be turned into an inequality constraint:

Simpson's Integration:

$$y_i(n+1) = y_i(n) + \frac{\tau}{6}[F_i(n) + 4F_i(n2) + F_i(n+1)] \pm \epsilon$$

Polynomial Interpolation:

$$y_i(n2) = \frac{1}{2}[y_i(n) + y_i(n+1)] + \frac{\tau}{8}[F_i(n) - F_i(n+1)] \pm \epsilon.$$

Here, care must be taken to ensure that ϵ is large enough to correct for the integration errors between integration techniques, but not so large as to overwhelm the underlying dynamics of this system.

7.2 Starting Guess

For complicated systems, the optimization is highly dependent on the starting guess. For simpler problems, such as the Hodgkin-Huxley and Colpitts examples, all state variables can be set to a constant for all time as the initial guess, resulting in an excellent parameter estimation. This is not the case for more complicated models.

For twin experiments, a common technique is start the optimization on the correct answer for all parameters and all state variables. For SQP solvers, such as SNOPT, this technique typically results in very fast convergence, due to the algorithmic details of the optimizer. For interior point methods, such as IPOPT, this does not necessarily work as well, for the same reason.

Starting on the correct answer is not possible for non-twin experiments, however, so other techniques are necessary. Since DSPE is based on the idea of synchronization of data to model, one method is to take the data and numerically synchronize the data to the model in an integration with fixed coupling constant. This integration will necessarily make some assumption about the values of the initial conditions and parameters, but as long as these are in the feasible range given to the optimization, the resulting time series gives a decent place to start.

Using this numerical synchronization method should not be limited to one set of parameters and initial conditions; a range of starting guesses for the optimization must be used to determine if DSPE converges to the same solution.

Chapter 8

Conclusion

Nonlinear parameter and state estimation is a challenging discipline and developing a method that is generally applicable is probably impossible. The dynamic state and parameter estimation method described here is a valuable tool in this broad field, and has shown successful results in a wide range of problems in diverse disciplines such as electrical circuits, oceanography, and neurobiology. Current efforts involve estimation of parameters and states in networks of neurons and experimental birdsong systems.

The development of DSPE Python scripts to simplify implementation of dynamic state and parameter estimation has greatly enhanced the utility of this method when presented with experimental data by allowing for easy testing of multiple models to fit a system. The implementation of larger and more complicated systems brings new challenges, as evidenced by the lobster lateral pyloric neuron model. Expansion to experimental systems brings challenges of its own, specifically the role of noise in both the dynamical model and the experimental data, and other methods such as the Monte Carlo path integral method may be needed.

The DSPE scripts, as well as the minAzero scripts, are valuable tools that are being implemented on a wide range of problems in biology, oceanography, and many other disciplines as the method matures. Accurate state and parameter estimation paves the way for accurate prediction, and the methods and software presented here make a valuable addition to the parameter estimation toolkit.

Appendix A

IP control User Manual

A.1 Introduction

This document describes the current version (2.1) of the IP control software package. It explains how to use the software I have developed from January 2009 - Present, which uses the IPOPT optimization package for optimal control tracking and dynamic parameter estimation of dynamic systems.

A.1.1 Problem Statement

Given a set of first-order differential equations in the state vector $x(t)$:

$$\frac{dx_1(t)}{dt} = G_1(x_1(t), x_\perp(t), q) \tag{A.1}$$

$$\frac{dx_\perp(t)}{dt} = G_\perp(x_1(t), x_\perp(t), q). \tag{A.2}$$

Determine unobserved states $\mathbf{x}_\perp(t)$ and parameters $\mathbf{q}$ of this system using a time-series observation of one (or more) $x_1(t)$ of the state variables.

This software transforms a text file representation of the dynamical system into an IPOPT C^{++} code that minimizes a user-provided cost function, subject to the differential equations and user-provided bounds on the parameters and state variables.

A.2 Installations

I will not go into detail about how to install the necessary software to make everything work, since this is definitely platform dependent. I have successfully installed and run on Mac OS, as well as Red Hat and Ubuntu LINUX, so I expect the software will run on most systems. The needed software is:

- Python version 2.5 or later. This should be available in most package managers, or download from source at www.python.org
- SymPy version 0.6.3 or later. This is a Python module that is used to perform symbolic differentiation, and is available at code.google.com/p/sympy/.
- IPOPT version 3.5.4 or later. This is available at https://projects.coin-or.org/Ipopt. Install the C^{++} version (which will require a C^{++} compiler - I use g++)

A.2.1 Python

Python and SymPy installations should be straightforward - run through some of the examples from the SymPy site to make sure the module is working correctly.

A.2.2 IPOPT

The IPOPT installation is a bit more complicated, but is well documented at http://www.coin-or.org/Ipopt/documentation/. Significantly, external linear solver codes are necessary for interaction with the IPOPT software - I have followed the instructions in the above documentation to install BLAS, LAPACK, ASL, MUMPS, MA27, and MC19 (the latter two are in the Download HSL Subroutines section), and Pardiso (requires user registration). As noted on the main IPOPT site, make sure to read the current issues page before installation - this is located at: https://projects.coin-or.org/BuildTools/wiki/current-issues.

After these installations, I strongly recommend running the make test for IPOPT. This will run through some examples from the directory root/.../build/Ipopt/examples.

If these examples work correctly, then my software should work correctly. Make note of this directory, since the makefiles for the examples in this directory will be necessary to consult to determine all the compiler flags for any given problem.

A.2.3 Scripts

The control package consists of seven python files. Once all the above software is installed, all that is needed is two additional, user-generated text files to generate C^{++} code for a new dynamical parameter estimation problem. The seven python files must be either in whichever directory that the problem will be run in, or (preferably) stashed in the directory where python modules typically go (I am not sure if this is standard across platforms). The trick that I have used is to put the six python files into a directory that is part of the PATH environmental variable and change the makecode.py script into an executable. Actual implementation of this depends on the flavor of UNIX/LINUX that is used, as well as the shell, so I will not provide details here.

A.2.4 Modifying makemake.py

The Makefile compiles C^{++} object files and links them with the installed IPOPT libraries, in order to create an executable. Since the location of the IPOPT libraries, as well as the flags used to compile them, differ between installations, this file will be unique to a given machine. Modification of the makemake.py script to give correct Makefiles for a given machine consists of:

- Ensure that the IPOPT installation proceeded correctly, as evidenced by zero errors for the "make install" step.
- In the IPOPT build directory, try to compile (make) one of the examples, for instance at /build/Ipopt/examples/hs071_cpp.
- If this compiles and runs correctly, open the Makefile in this directory.
- Make note of the entries in the following fields of this Makefile: CXX, CXXFLAGS, CXXLINKFLAGS, prefix, LIBS.

- In makemake.py, replace the default entries for these fields with those given in the example Makefile.
  - makemake.py is formatted differently than a Makefile, since it is a python code generation script.
  - Lines that begin with the '#' sign will be comments in the Makefile - leave these alone.
  - All lines must end with \n\ in order for the Makefile to be generated correctly.
  - The best way to ensure that all the compile flags are correct is to copy and paste from the example Makefile, ensuring that the end line characters are in place.
- The modification of makemake.py must only be done once for a given machine, unless IPOPT is reinstalled for whatever reason.

A.3 Usage

Usage of the code is deceptively simple. As currently configured, the only python file that needs to be altered is makemake.py (discussed above). For any new problem, all that is needed are two text files and an optional functions file:

- equations.txt specifies the name of the problem, the number of variables, parameters, controls, stimuli, and functions, the vector field, the objective function, and the names of the variables, parameters, controls, stimuli, and functions. For a given problem, equations.txt sets up the vector field, Jacobian, and Hessian.
- specs.txt specifies all the actual run parameters that may change for a given problem. This includes the timestep of the data, length of data set, names of the requisite data files, initial and boundary conditions for all variables, parameters, and controls, and how much tolerance to put on the equality constraints.

- myfunctions.cpp is a C^{++} file that defines functions used in equations.txt.

A.3.1 Equations.txt

This text file is setup as follows:

- Line 1: Problem Name.
- Line 2: nY, nP, nU, nI, nF.
  - nY = Number of model dynamical variables.
  - nP = Number of model unknown parameters.
  - nU = Number of controls to couple data to model dynamical variables.
  - nI = Number of input stimuli to the model.
  - nF = number of functions used to simplify the model representation.
- Lines 3→2+nY: Vector field of the dynamical model. Each line is the right-hand side of the equation $\frac{dx_i(t)}{dt} = G_i(\mathbf{x}(t), q)$.
- Line 3+nY: Objective function.
- Lines 4+nY→3+2*nY: Dynamical variable names, given in the same order as the vector field equations.
- Lines 4+2*nY→3+2*nY+nP: Unknown parameter names.
- Lines 4+2*nY+nP→3+2*nY+nP+nU: Control names.
- Lines 4+2*nY+nP+nU→3+2*nY+nP+2*nU: Names given to coupled data sets. Each control must include one data variable.
- Lines 4+2*nY+nP+2*nU→3+2*nY+nP+2*nU+nI: Names of input stimuli.
- Lines 4+2*nY+nP+2*nU+nI→3+2*nY+nP+2*nU+nI+nF: Names of any functions used, followed by a comma and the number of arguments in the function. The name of the function must be identical to the name given in a provided myfunctions.cpp file, discussed below.

Any changes to the equations.txt file will require running makecode.py and make in order to take effect.

A.3.2 Specs.txt

This text file is loaded by the final executable program, and is setup as follows:

- Line 1: Problem Length T. Since the discretization method used involves data midpoints, the total number of data points used is 2*T+1.
- Line 2: skip = How many lines to skip at the beginning of a given data file. This allows for sampling over different parts of a given data set.
- Line 3: Time step of the data. Again, since midpoints are used in the discretization, this is twice the time step used to generate the data.
- Lines 4→3+nU: Data file names for coupled data. Each coupled data variable must have its own data file, listed in the order of the respective controls given in equations.txt. A data file of this name must be included in the directory of the executable and contain at least as many lines as given by 2*T+1+skip, or the program will give a segmentation fault. Format of the data file is one number (integer or decimal) per line.
- Lines 4+nU→3+nU+nI: Data file names for input stimuli. Same rules as for data files for coupled data.
- Line 4+nU+nI: 0 or 1. This determines whether a data file will be used to give an initial guess for the optimization for all state variables at all time points. 0 means NO, 1 means YES.
  - IF 0: Line 5+nU+nI: See below.
  - IF 1: Line 5+nU+nI: Data file name for initial optimization. Each line of this data file contains a value for each state variable at one time point, with either space or tab delimiters.

- Lines 5(6)+nU+nI→4(5)+nU+nI+nY: Bounds, initial guess, and constraint variability (optional) for each of the state variables. Each value is separated by a comma, in the order, lower bound, upper bound, initial guess, variability.
  - The bounds are valid for the dynamical variable for each time point.
  - The initial guess is valid for the dynamical variable for each time point if no initial data file is used.
  - The variability, δ, converts the Hermite Simpson integration from equality constraints to inequality constraints in order to account for numerical instabilities within the integration and within the vector field. Instead of x(n+1) = x(n) + Δ*F(n), where F(n) is the Hermite integration rule, the variability option gives, x(n+1) = x(n) + Δ*F(n) ± δ
- Lines 5(6)+nU+nI+nY→4(5)+nU+nI+nY+2*nU: Bounds and initial guess for the control and derivative of the control. Each control is followed by its own derivative.
- Lines 5(6)+nU+nI+nY+2*nU→4(5)+nU+nI+nY+2*nU+nP: Bounds and initial guess for the unknown parameters.

Changes to specs.txt do not require recompilation of either makecode.py or make.

A.3.3 Myfunctions.cpp

This C^{++} file defines any functions used in equations.txt. The function, its derivatives with respect to each variable, and 2^{nd} derivatives with respect to each combination of variables must be defined as depicted in the example provided. The function name *func* must be as given in equations.txt, and the 1^{st} and 2^{nd} derivatives by *funcjac* and *funches*.

A.3.4 Makecode.py

Given a properly formatted equations.txt, the script makecode.py will generate a unique set of C^{++} files to run an IPOPT program. If all the python scripts

are in the directory with equations.txt, run the script with the command, "python makecode.py". If the makecode.py script has been turned into an executable, and all the scripts stored somewhere in the PATH environmental variable, then the command "makecode.py" suffices. Upon this command, the following occurs:

- The code discretize.py takes the equations.txt file as input, and generates a symbolic vector field using python's symbolic math module, SymPy. Discretize.py then takes this vector field and discretizes the integration according to Simpson's rule. Finally, the derivatives and second derivatives of the vector field are taken with respect to all variables, parameters, and controls. All non-zero entries are stored in row, column, value format (with some additional information) in Jacobian and Hessian arrays. Five separate arrays from discretize.py are needed in the main IPOPT program. These are the objective function, the constraints, the gradient of the objective, the Jacobian of the constraints, and the Hessian of the objective and constraints.

- The C^{++} code that IPOPT runs is generated as follows:

  - Makecpp.py generates a "problemname_main.cpp" file that defines a new problem and includes all the appropriate IPOPT calls.

  - Makehpp.py generates a "problemname_nlp.hpp" file that serves as the header file for the problem class.

  - Makecode.py generates a "problemname_nlp.cpp" file that defines the problem class. This problem class sets up the problem (problem size, bounds, initial conditions) and defines the cost function, constraints, Jacobian, Hessian, and output files.

  - Makemake.py generates a Makefile that links these files together with the IPOPT libraries - this is the one file that will have to change depending on the platform.

  - Makeopt.py generates a "problemname.opt" option file for the problem.

  - Correlate.py is used to generate one of the output files, Rvalue.dat.

- All five of these files are generated by makecode.py - the other python scripts are invoked from within this module. These scripts open new files, based on the problem name given in equations.txt, and write appropriate character strings to these files to turn them into C^{++} files. The main algorithmic parts are in the makecode.py module - mostly due to modifying the discretized equations into loops over all time points, with variables, parameters, and controls at each time step pointing to the correct spot in the optimization variable arrays for the constraints, Jacobian elements, and Hessian elements.

- The makecode.py script will generate the needed C^{++} files in a matter of seconds for small (3 state variables) problems, but can take considerably longer for higher dimensionality due to the increased number of derivatives that need to be done. This will generate the needed C^{++} files which are linked and compiled by a Makefile. Provided that the Makefile is made appropriately (as stated previously, this is platform dependent, and should be checked relative to the make check of IPOPT), all that is needed is a make (compile), and then run of the problem.

A.3.5 Running the Program

After the makecode.py and make steps, an executable program will be created, "problemname_cpp". Ensure that specs.txt is constructed correctly and all data files specified are in the working directory. The "problemname.opt" option file can be amended to to give specific IPOPT run options (e.g., termination criteria, linear solver choice). Then run the executable program.

A.3.6 Advanced Usage

Supported problem types include multiple controls, injected stimuli (e.g., current), and setting initial conditions for all state variables at all discretized times points. These are shown in the given examples, but briefly:

- Multiple controls are set in the equations.txt file with the nU variable and addition of the controls to the actual equations. In the specs.txt file where

the range and initial value are set, each control is followed by its associated differential.

- Injected stimuli are used by including the stimuli in equations.txt, and including the name of the injected stimuli data file in specs.txt. This data file is of the same format as the data file for the measured data file - one value per line, and should be the same number of lines for a given problem.

- Initial conditions (really an initial guess) are set in the specs.txt file. A binary toggle determines whether a data file will be used to set an initial guess for all state variables. When set to zero, the initial guess for all variables, controls, and parameters is set as a constant for all time steps according to the third column of the appropriate row in the specs.txt file (see examples). If the toggle is set to 1, the subsequent uncommented line of specs.txt must contain the name of the data file with the initial guess. The hh example in the example directories includes this data file (initial.dat), for instance. The initial.dat file contains a column for each state variable per row, with the same number of rows as the measured data file.

A.4 Output

The output from running a dynamical control problem will generate five data files.

- **Ipopt.out** gives details about the optimization iterations.

- **Param.dat** lists the parameter values in the order given in equations.txt, one per line.

- **Data.dat** lists the variables and controls at each time step. The format of data.dat is: counter, x1, x2, ..., xn, u1, u2, ..., uk, followed by the y1, ..., yk of the input data.

- **Rvalue.dat** lists the Rvalue at each time point. The Rvalue is a measure of the relative contribution of the model dynamics and control terms in the

output, and gives a measure of the quality of the state estimation.

- **Carl.input** outputs the state variables and parameters in the format needed for input into the cudaPIMC code.

A.5 Troubleshooting

I have tested these scripts over a wide range of problems, so I believe that the algorithms are correct. However, there are a few common errors that may crop up.

- Variable and parameter naming is very important. At few common problems can crop up. Never use a variable name that includes the name of another variable. For instance p1 and p11 would be bad, since p11 includes p1. In this case, p01 and p11 would be adequate. Along this vein, all variable names should be at least 2 characters long, just in case.

- Another naming problem is associated with an internal Python/sympy function called sympify. This function is used to transform python arrays into C^{++} code, but it can be somewhat finicky with certain variable names. As an example, the name "gamma" in equations.txt for a parameter name will results in an error message upon running makecode.py which includes something like: TypeError: bad operand type for unary -: 'FunctionClass'. I do not have a complete list of problematic names, but be aware of this potential error.

- Another common mistake is to neglect to include the proper data files in the working directory. These files must have names exactly as given in the specs.txt file, or else execution of the program will result in a bus error or segmentation fault. If these files have the correct name, but improper format (e.g., not long enough), these types of memory errors may occur as well.

- One further error occurs if equations.txt and specs.txt are not saved as pure text files. An additional (invisible) character string may appear, uncommented, in the files, thus disrupting the read statements that occur in python

and C^{++}. As a quick check, open these files in a gui text editor, and then save them as plain text to avoid this problem. This should not be a problem with the typical UNIX/LINUX command line text editors.

Appendix B

IP control code

B.1 Discretize.py

```
#########################################################
#
#  20 October 2009
#  Bryan A. Toth
#  University of California, San Diego
#  btoth@physics.ucsd.edu
#
#  This script performs symbolic Hermite-Simpson
#  integration on a vector field given in the text file
#  equations.txt, takes the Jacobian and Hessian of the
#  vector field, and stores the results in arrays that
#  are used by other python scripts in this directory.
#
#  This script has been developed as part of a suite of
#  python scripts to define a dynamic parameter estimation
#  problem using the optimization software IPOPT, but is
#  generally applicable to any application needing
#  discretized derivatives of a vector field.
#
#########################################################

import sympy as sym
from sympy import *
import re

#  Opening and reading the text file with vector field information
```

```
file = open('equations.txt','r')
temp=[]  # Array to hold equations.txt information
for line in file:
  if line.startswith('#'): # Pound used as comment in text file
     continue
  elif line.startswith('\\'): # In case file has UTF-8 markers
     continue
  else:
     temp.append(line)
file.close()

h=[]  # Array to hold unformatted equations.txt information
for i in range(len(temp)):
  temp1=temp[i].rstrip( )
  h.append(temp1)

# Initialize problem variables
nY=0
nP=0
nU=0
nI=0
nF=0

# Problem name
Problem = h[0]

# Problem variables
a=h[1].split(',')
nY=int(a[0])
nP=int(a[1])
nU=int(a[2])
nI=int(a[3])
if len(a) > 4:
  nF=int(a[4])

# Import equations as strings
Feqnstr = []
for k in range(nY):
   Feqnstr.append(h[k+2])

# Import objective function as string
Fobjstr = []
```

```
Fobjstr.append(h[nY+2])

# Import variable, parameter, control, data, and stimuli names
# as strings
Lvars = []
for k in range(nY):
   Lvars.append(h[k+3+nY])

Lparams = []
for k in range(nP):
   Lparams.append(h[k+3+nY+nY])

Lcouple = []
Ldata = []
for k in range(nU):
    Lcouple.append(h[k+3+nY+nY+nP])
    Lcouple.append('d'+ h[k+3+nY+nY+nP])
    Ldata.append(h[k+3+nY+nY+nP+nU])
Lstimuli = []
for k in range(nI):
    Lstimuli.append(h[k+3+2*nY+nP+2*nU])

# Import function names as strings
Funcstr = []
Funcarg = []
for k in range(nF):
    temp = h[k+3+2*nY+nP+2*nU+nI].split(',')
    Funcstr.append(temp[0])
    Funcarg.append(int(temp[1]))
#Lvars.reverse()
#Lcouple.reverse()
Fdim = len(Feqnstr)
Pdim = len(Lparams)

#  Make symbols using sympy module
Sv = []
Sp = []
Sk = []
Sd = []
Si = []
for i in range(len(Lvars)):
  Sv.append(sym.Symbol(Lvars[i]))
for i in range(len(Lparams)):
```

```
  Sp.append(sym.Symbol(Lparams[i]))
for i in range(nU):
  Sd.append(sym.Symbol(Ldata[i]))
  Sk.append(sym.Symbol(Lcouple[2*i]))
  Sk.append(sym.Symbol(Lcouple[2*i+1]))
# Sk includes coupling and derivative of the coupling: k1,k1d,etc ...
for i in range(nI):
  Si.append(sym.Symbol(Lstimuli[i]))

Sall = Sv + Sk + Sp

# Make symbols for functions
Sf = []
for i in range(nF):
  Sf.append(sym.Function(Funcstr[i]))

hstep = sym.Symbol("hstep")

# Define symbolic vector field
Feqns = []
for k in range(Fdim):
  sTemp1 = Feqnstr[k]
  for i in range(len(Lvars)):
    sTemp2 = "Sv[%d]" % i
    sTemp1 = sTemp1.replace(Lvars[i],sTemp2)
  for i in range(len(Lparams)):
    sTemp2 = "Sp[%d]" % i
    sTemp1 = sTemp1.replace(Lparams[i],sTemp2)
  for i in range(len(Lcouple)):
    sTemp2 = "Sk[%d]" % i
    sTemp1 = sTemp1.replace(Lcouple[i],sTemp2)
  for i in range(nU):
    sTemp2 = "Sd[%d]" % i
    sTemp1 = sTemp1.replace(Ldata[i],sTemp2)
  for i in range(nI):
    sTemp2 = "Si[%d]" % i
    sTemp1 = sTemp1.replace(Lstimuli[i],sTemp2)
  for i in range(nF):
    sTemp2 = "Sf[%d]" % i
    sTemp1 = sTemp1.replace(Funcstr[i],sTemp2)
  sTemp2 = "Feqns.append("
  sTemp2 = sTemp2 + sTemp1 + ")"
  exec sTemp2
```

```
# Define symbolic objective function
Fobj = []
sTemp1 = Fobjstr[0]
for i in range(len(Lvars)):
  sTemp2 = "Sv[%d]" % i
  sTemp1 = sTemp1.replace(Lvars[i],sTemp2)
for i in range(len(Lparams)):
  sTemp2 = "Sp[%d]" % i
  sTemp1 = sTemp1.replace(Lparams[i],sTemp2)
for i in range(len(Lcouple)):
  sTemp2 = "Sk[%d]" % i
  sTemp1 = sTemp1.replace(Lcouple[i],sTemp2)
for i in range(nU):
  sTemp2 = "Sd[%d]" % i
  sTemp1 = sTemp1.replace(Ldata[i],sTemp2)
sTemp2 = "Fobj.append("
sTemp2 = sTemp2 + sTemp1 + ")"
exec sTemp2


# For a continuous version of the Jacobian and Hessian of the
#  vector field, the following can be printed.  Otherwise, these
#  are not needed in the script.

#--------Jacobian---------------
#J = diff(Feqns,Sall[:]) # Jacobian
#---------Hessian---------------
#H = diff(J,Sall[:]) # Hessian


# Perform Hermite-Simpson discretization and label these as
#  constraints in the optimization problem.

#  The format of these constraints is one element in AllCon for
#   each dynamical variable in the vector field, with each entry
#   containing three entries for the discretization, corresponding
#   to the current time-step, next time-step, and midpoint value,
#   as defined by the integration rule.  For other integration rules,
#   this part must change.
AllCon = []
# Simpson Constraints
for k in range(len(Sv)):
```

```
  tCon = []
  tCon.append(Sv[k] + (hstep/6.0)*Feqns[k])
  tCon.append(-Sv[k] + (hstep/6.0)*Feqns[k])
  tCon.append((2.0*hstep/3.0)*Feqns[k])
  AllCon.append(tCon)

# Hermite Constraints
for k in range(len(Sv)):
  tCon = []
  tCon.append(0.5*Sv[k] + (hstep/8.0)*Feqns[k])
  tCon.append(0.5*Sv[k] - (hstep/8.0)*Feqns[k])
  tCon.append(-Sv[k])
  AllCon.append(tCon)

# Hermite Control Constraint
for k in range(nU):
  tCon = []
  tCon.append(0.5*Sk[2*k] + (hstep/8.0)*Sk[2*k+1])
  tCon.append(0.5*Sk[2*k] - (hstep/8.0)*Sk[2*k+1])
  tCon.append(-Sk[2*k])
  AllCon.append(tCon)

# Add objective function
AllObj = []
AllObj.append(Fobj[0])
AllObj.append(0)
AllObj.append(Fobj[0])

#  The following dictionaries are for a subsequent substitution
#   in the function subvars.  These are the actual variable array
#   names that will be used in the eventual c++ IPOPT program.


dict1 = {0:"Xval",1:"Xvalp1",2:"Xval2"}
dict2 = {0:"K11val",1:"K11valp1",2:"K11val2"}
dict3 = {0:"dK11val",1:"dK11valp1",2:""}
dict4 = {0:"Xdval",1:"Xdvalp1",2:"Xdval2"}
dict5 = {0:"Ival",1:"Ivalp1",2:"Ival2"}

#  The following function performs a variable substitution, and
#   is called later in the code.

def subvars(mystr,myi):
```

```
   mytemp = mystr
#  The following two lines are very important
#  Sympy converts all inputs into a simplified form
#  for calculations.  Specifically, this involves
#  any exponentials (a^b) put in the form a**b.
#  Whereas this form is acceptable for fortran outputs,
#  this needs to change to pow(a,b) for C++ outputs.
#  Sympify and ccode combine to make this transformation.
#  This transformation is done at this point in the code,
#  since sympify will not operate on an
#  expression that includes brackets - which are added
#  in this function.
   mytemp = sym.sympify(mytemp)
   mytemp = sym.ccode(mytemp)

   for j in range(len(Sv)):
#      Srep = dict1[n] + "[%d]" % (len(Sv)-j-1)
      Srep = dict1[n] + "[%d]" % j
      Sfind = Lvars[j]
      mytemp = mytemp.replace(Sfind,Srep)

   for j in range(len(Sp)):
      Srep = "Pval[%d]" % j
      Sfind = Lparams[j]
      mytemp = mytemp.replace(Sfind,Srep)

#   for j in range(len(Sk)):
#      Srep = dict2[n] + "[%d]" % (j/2)
#      Sfind = Lcouple[j]
#      mytemp = mytemp.replace(Sfind,Srep)
   for j in range(len(Sk)):
      if (j % 2 == 0):
#        Srep = dict2[n] + "[%d]" % ((len(Sk)-j-1)/2)
        Srep = dict2[n] + "[%d]" % (j/2)
        Sfind = Lcouple[j]
        mytemp = mytemp.replace(Sfind,Srep)

      elif (j % 2 == 1):
#        Srep = dict3[n] + "[%d]" % ((len(Sk)-j-1)/2)
        Srep = dict3[n] + "[%d]" % (j/2)
        Sfind = Lcouple[j]
        mytemp = mytemp.replace(Sfind,Srep)
```

```
   for j in range(len(Sd)):
      Srep = dict4[n] + "[%d]" % j
      Sfind = Ldata[j]
      mytemp = mytemp.replace(Sfind,Srep)

   for j in range(len(Si)):
      Srep = dict5[n] + "[%d]" % j
      Sfind = Lstimuli[j]
      mytemp = mytemp.replace(Sfind,Srep)

   return mytemp
# END subvars

def subfunc(mystr,myi):
   mytemp = mystr
   Dsearch = re.findall('D\(', mytemp)
   for num in range(len(Dsearch)):
     jacsearch = re.search('D\(([a-z]+)\((-?[0-9]+(\.[0-9]+)?,
     |[A-Za-z0-9]+\[[0-9]+\], )+(-?[0-9]+(\.[0-9]+)?
     |[A-Za-z0-9]+\[[0-9]+\])\), (-?[0-9]+(\.[0-9]+)?
     |[A-Za-z0-9]+\[[0-9]+\])\)', mytemp)
     hessearch = re.search('D\(([a-z]+)\((-?[0-9]+(\.[0-9]+)?,
     |[A-Za-z0-9]+\[[0-9]+\], )+(-?[0-9]+(\.[0-9]+)?
     |[A-Za-z0-9]+\[[0-9]+\])\), (-?[0-9]+(\.[0-9]+)?
     |[A-Za-z0-9]+\[[0-9]+\]), (-?[0-9]+(\.[0-9]+)?
     |[A-Za-z0-9]+\[[0-9]+\])\)', mytemp)
     if jacsearch:
       dervar = jacsearch.group(6)
       var = re.findall('[A-za-z0-9]+\[[0-9]+\]|-?
       [0-9]*\.?[0-9]+', jacsearch.group(0))
       for i in range(len(var)-1):
         if var[i] == dervar:
           jacnum = i+1
       rep = jacsearch.group(1) + 'jac('
       for i in range(len(var)-1):
         temp = var[i] + ','
         rep += temp
       rep += str(jacnum) +')'
       mytemp = re.sub('D\(([a-z]+)\((-?[0-9]+(\.[0-9]+)?,
        |[A-Za-z0-9]+\[[0-9]+\], )+(-?[0-9]+(\.[0-9]+)?
        |[A-Za-z0-9]+\[[0-9]+\])\), (-?[0-9]+(\.[0-9]+)?
        |[A-Za-z0-9]+\[[0-9]+\])\)', rep,mytemp,1)
     if hessearch:
```

```
        dervar1 = hessearch.group(6)
        dervar2 = hessearch.group(8)
        hesnum1 = 0
        hesnum2 = 0
        var = re.findall('[A-za-z0-9]+\[[0-9]+\]|-?
        [0-9]*\.?[0-9]+', hessearch.group(0))
        for i in range(len(var)-2):
          if var[i] == dervar1:
            hesnum1 = i+1
          if var[i] == dervar2:
            hesnum2 = i+1
        rep = hessearch.group(1) + 'hes('
        for i in range(len(var)-2):
          temp = var[i] + ','
          rep += temp
        rep += str(hesnum1) +','+ str(hesnum2)+ ')'
        mytemp = re.sub('D\(([a-z]+)\((-?[0-9]+(\.[0-9]+)?,
        |[A-Za-z0-9]+\[[0-9]+\], )+(-?[0-9]+(\.[0-9]+)?
        |[A-Za-z0-9]+\[[0-9]+\])\), (-?[0-9]+(\.[0-9]+)?
        |[A-Za-z0-9]+\[[0-9]+\]), (-?[0-9]+(\.[0-9]+)?
        |[A-Za-z0-9]+\[[0-9]+\])\)', rep, mytemp, 1)
    return mytemp
#end subfunc
# Build constraint equation strings
strAllCon = []
for icon in range(len(AllCon)):
    temp1 = []
    for n in [0,1,2]:
       Stemp = str(AllCon[icon][n])
       Stemp = subvars(Stemp,n)
       Stemp = subfunc(Stemp,n)
       temp1.append(Stemp)
    strAllCon.append(temp1)
# Build objective function string
strObj = []
temp1 = []
for n in [0,1,2]:
     Stemp = str(AllObj[n])
     Stemp = subvars(Stemp,n)
     Stemp = subfunc(Stemp,n)
     temp1.append(Stemp)
strObj.append(temp1)
```

```
# Build Jacobian strings
sJac = []
for icon in range(len(AllCon)):
   temp1 = []
   for jvar in range(len(Sall)):
      temp2 = []
      for n in [0,1,2]:
         Stemp = str(sym.diff(AllCon[icon][n],Sall[jvar]))
         Stemp = subvars(Stemp,n)
 Stemp = subfunc(Stemp,n)
         temp2.append(Stemp)
      temp1.append(temp2)
   sJac.append(temp1)
# Build Objective gradient strings
sObj = []
for jvar in range(len(Sall)):
   temp2 = []
   for n in [0,1,2]:
      Stemp = str(sym.diff(AllObj[n],Sall[jvar]))
      Stemp = subvars(Stemp,n)
      Stemp = subfunc(Stemp,n)
      temp2.append(Stemp)
   sObj.append(temp2)

# Build Hessian strings
sHes = []
for icon in range(len(AllCon)):
   temp1 = []
   for jvar in range(len(Sall)):
      temp2 = []
      for kvar in range(len(Sall)):
         temp3 = []
         for n in [0,1,2]:
    Stemp = str(sym.diff(sym.diff(AllCon[icon][n],Sall[jvar]),
    Sall[kvar]))
            Stemp = subvars(Stemp,n)
    Stemp = subfunc(Stemp,n)
            temp3.append(Stemp)
         temp2.append(temp3)
      temp1.append(temp2)
   sHes.append(temp1)

# Add Hessian for the objective function
```

```
temp1=[]
for jvar in range(len(Sall)):
    temp2 = []
    for kvar in range(len(Sall)):
temp3 = []
for n in [0,1,2]:
    Stemp = str(sym.diff(sym.diff(AllObj[n],Sall[jvar]),Sall[kvar]))
    Stemp = subvars(Stemp,n)
    Stemp = subfunc(Stemp,n)
    temp3.append(Stemp)
temp2.append(temp3)
    temp1.append(temp2)
sHes.append(temp1)

# Fill out Jacobian vector
# Jacobian includes all constraints, but not objective
# VJac will include all non-zero elements of the Jacobian
#  with the format: (value, row, column, time) where time
#  refers to whether the element is current time, next time
#  or midpoint time in discretized form

VJac = []
for i in range(len(AllCon)):
   for j in range(len(Sall)):
         if j < (len(Sv) + len(Sk)): # Same as nY + 2*nU
             for n in [0,1,2]:
                F = sJac[i][j][n]
                if F != '0':
                    temp1 = []
                    temp1.append(F)
                    temp1.append(i)
                    temp1.append(j)
                    temp1.append(n)
                    VJac.append(temp1)
# Distinction is made between variable/control entries and parameter
#  entries based on how the sJac matrix is set up: derivatives with
#  respect to the parameters must take into account all time
#  information-adding the current time, next time, and midpoint time
#  derivatives together into a single entry.

         else:
             F = sJac[i][j][0]
             if F != '0':
```

```
                temp2 = []
                temp1 = ''
                for n in [0,1,2]:
                    F = sJac[i][j][n]
                    if n == 0:
temp1 = temp1 + F
            else:
                        temp1 = temp1 + '+' + F
                temp2.append(temp1)
        temp2.append(i)
         temp2.append(j)
                VJac.append(temp2)

# Fill out objective gradient
VObj = []
for j in range(len(Sall)):
    if j < (len(Sv) + len(Sk)):
        for n in [0,1,2]:
            F = sObj[j][n]
            if F != '0':
                temp1 = []
                temp1.append(F)
                temp1.append(j)
                temp1.append(n)
                VObj.append(temp1)
    else:
        F = sObj[j][0]
        if F != '0':
            temp2 = []
            temp1 = ''
            for n in [0,1,2]:
                F = sObj[j][n]
                if n == 0:
                    temp1 = temp1 + F
                else:
                    temp1 = temp1 + '+' + F
            temp2.append(temp1)
            temp2.append(j)
            VObj.append(temp2)

# Fill out Hessian vector
# This vector has the format [index, constraint, row, column, time,
#  value] for each entry
```

```
#  Index is a counter that refers to a specific row/column
#  combination,so that the same combination is not used more
#  than once.  Constraint refers to which constraint that the
#  Hessian element is taking derivatives of

VHes = []

index = 0
oddball = 0

# index tracks how many entries there are, oddball tracks the
# parameter/parameter entries.  This distinction is necessary since
# each time step will have its own entries, but parameter/parameter
# derivatives only have one Hessian entry.

# Fill out for constraint 1 first
# Note that this is a symmetrical matrix - filling out lower
#  diagonal-half only.

for j in range(len(Sall)):
# Sall is Sv + Sk + Sp, the symbolic representations of the
# variables, controls, and parameters.
   for k in range(j+1):
     if k < (len(Sv) + len(Sk)):
       for n in [0,2]:
         H = sHes[0][j][k][n]
         if H != '0':
             temp1 = []
             temp1.append(index)
             temp1.append(0)
             temp1.append(j)
             temp1.append(k)
             temp1.append(n)
             temp1.append(H)
     temp1.append(1)
     if n == 0:
temp1.append(sHes[0][j][k][1])
             VHes.append(temp1)
             index = index + 1
     else:  # These are the parameter/parameter derivatives
       H = sHes[0][j][k][0]
       if H != '0':
         temp2 = []
```

```
 temp1 = ''
         for n in [0,1,2]:
           H = sHes[0][j][k][n]
   if n == 0:
      temp1 = temp1 + H
   else:
      temp1 = temp1 + '+' + H
         temp2.append(index)
 temp2.append(0)
 temp2.append(j)
 temp2.append(k)
 temp2.append(-1)
         temp2.append(temp1)
 temp2.append(1)
 VHes.append(temp2)
 index = index + 1
 oddball = oddball + 1


# Fill out additional constraints, checking to see if row/column
# has been indexed already.  Need to make distinction for
# constraint/constraint derivatives
for i in range(len(AllCon)):
   for j in range(len(Sall)):
      for k in range(j+1):
        if k < (len(Sv) + len(Sk)):
          for n in [0,2]:
            H = sHes[i+1][j][k][n]
            new = 0
            if H != '0':
        # Check to see if row/column has been used before
                for u in range(len(VHes)):
                     if j == VHes[u][2]:
                         if k == VHes[u][3]:
                           if n == VHes[u][4]:
       oldindex = VHes[u][0]
         new = 1
                                temp1 = []
                                temp1.append(oldindex)
                                temp1.append(i+1)
                                temp1.append(j)
                                temp1.append(k)
                                temp1.append(n)
```

```
                                temp1.append(H)
       temp1.append(0)
                                if n == 0:
temp1.append(sHes[i+1][j][k][1])
                if new == 1:
                   VHes.append(temp1) # Use old index
                elif new == 0: # Assign new index to new row/column
                                          # combination
                   temp1 = []
                   temp1.append(index)
                   temp1.append(i+1)
                   temp1.append(j)
                   temp1.append(k)
                   temp1.append(n)
                   temp1.append(H)
   temp1.append(1)
   if n==0:
    temp1.append(sHes[i+1][j][k][1])
                   VHes.append(temp1)
                   index = index + 1
        else: # Take care of the oddballs: parameter/parameter
                  #  combinations
  H = sHes[i+1][j][k][0]
  new = 0
  if H != '0':
    for u in range(len(VHes)):
if j == VHes[u][2]:
    if k == VHes[u][3]:
oldindex = VHes[u][0]
new = 1
temp2 = []
temp1 = ''
for n in [0,1,2]:
    H = sHes[i+1][j][k][n]
    if n ==0:
temp1 = temp1 + H
    else:
temp1 = temp1 + '+' + H
temp2.append(oldindex)
temp2.append(i+1)
temp2.append(j)
temp2.append(k)
temp2.append(-1)
```

```
temp2.append(temp1)
temp2.append(0)
    if new == 1:
VHes.append(temp2)
    elif new == 0:
temp2 = []
temp1 = ''
for n in [0,1,2]:
    H = sHes[i+1][j][k][n]
    if n == 0:
temp1 = temp1 + H
    else:
temp1 = temp1 + '+' + H
temp2.append(index)
temp2.append(i+1)
temp2.append(j)
temp2.append(k)
temp2.append(-1)
temp2.append(temp1)
temp2.append(1)
VHes.append(temp2)
index = index + 1
oddball = oddball + 1
```

B.2 Makecode.py

```
#!/usr/bin/python

#########################################################
#
#  20 October 2009
#  Bryan A. Toth
#  University of California, San Diego
#  btoth@physics.ucsd.edu
#
#  This script builds C++ code to run a dynamical parameter
#   estimation optimization problem with the optimization
#   software IPOPT.
#
#  Specifically, given a vector-field (model) of the form:
#
```

```
#           dx_1(t) = G_1(x_1(t),x_p(t),q)
#
#           dx_p(t) = G_p(x_1(t),x_p(t),q)
#
#          where x_p denotes 1 or more equations in the model,
#
#  this code takes the discretized vector field and objective
#  function (discretized in companion script discretize.py),
#  and builds the requisite IPOPT functions to solve the
#  resulting optimization problem.
#
#  This script has been developed as part of a suite of
#  python scripts to define a dynamic parameter estimation
#  problem using the optimization software IPOPT, but could
#  easily be modified for use with other optimization software.
#
#########################################################


#  For ease of use, all necessary C++ files are built with one
#   keyboard command.  The following four scripts each write a
#   necessary C++ file. See the individual scripts for more
#   information.


import discretize

import correlate
import makecpp
import makehpp
import makemake
import makeopt
# Discretize.py reads the file equations.txt and sets up the given
#  vector field in the correct format.

# Import the problem name and change to upper and lower case
prob = discretize.Problem
probu = prob.upper()
probl = prob.lower()

FILE = probl + '_nlp.cpp'

nU = discretize.nU
```

```
nP = discretize.nP
nY = discretize.nY
nI = discretize.nI
nF = discretize.nF

# The name of the IPOPT file to be written to
f = open(FILE,'w')

# The following write commands are writing C++ code.

# Front matter
f.write(`// %s.cpp\n' % probl)
f.write(`// Nonlinear Ipopt program\n')
f.write(`\n// Author: Bryan A. Toth\n// btoth@physics.ucsd.edu\n\n')
f.write(`#include \"%s_nlp.hpp\"\n' % probl)
f.write(`#include <cmath>\n\
#include <cstdio>\n\
#include <iostream>\n\
#include <fstream>\n\
#include <string>\n\
#include <stdlib.h>\n\
#include <cstring>\n')

f.write(`#ifdef HAVE_CSTDIO\n\
# include <cstdio>\n\
# include <iostream>\n\
# include <fstream>\n\
# include <string>\n\
# include <stdlib.h>\n\
#else\n\
# ifdef HAVE_STDIO_H\n\
#  include <stdio.h>\n\
# else\n\
#  error \"don\'t have header file for stdio\"\n\
# endif\n\
#endif\n\n')

f.write(`using namespace Ipopt;\n\n')
f.write(`using namespace std;\n\n')
for i in range(nF):
  args = discretize.Funcarg[i]
  f.write(`double %s(' % discretize.Sf[i])
  for j in range(args-1):
```

```
      f.write(‘double, ’)
   f.write(‘double);\n’)
   f.write(‘double %sjac(’ % discretize.Sf[i])
   for j in range(args-1):
      f.write(‘double, ’)
   f.write(‘double, int);\n’)
   f.write(‘double %shes(’ % discretize.Sf[i])
   for j in range(args-1):
      f.write(‘double, ’)
   f.write(‘double, int, int);\n’)
f.write(‘// constructor\n\
%s_NLP::%s_NLP()\n\
{\n’ % (probu, probu))

# Define problem parameters
f.write(‘  nU=%d;\n’ % nU)
f.write(‘  nP=%d;\n’ % nP)
f.write(‘  nY=%d;\n’ % nY)
f.write(‘  nI=%d;\n\n’ % nI)

# Define variable names
f.write(‘\
     K11val = new double[nU];\n\
     K11val2 = new double[nU];\n\
     K11valp1 = new double[nU];\n\
     dK11val = new double[nU];\n\
     dK11val2 = new double[nU];\n\
     dK11valp1 = new double[nU];\n\
     Xdval = new double[nU];\n\
     Xdval2 = new double[nU];\n\
     Xdvalp1 = new double[nU];\n’)
f.write(‘\
     Xval = new double[nY];\n\
     Xval2 = new double[nY];\n\
     Xvalp1 = new double[nY];\n’)
f.write(‘\
     Pval = new double[nP];\n’)
f.write(‘\
     Ival = new double[nI];\n\
     Ival2 = new double[nI];\n\
     Ivalp1 = new double[nI];\n\
     Rvalue = new double[nU];\n’)
# Commands to read specs.txt file
```

```
f.write(‘\
     string buffer;\n\
     specs = new string[6+nP+nY+nI+3*nU];\n\
     \n\
     int count;\n\
     count = 0;\n\
     \n\
     ifstream fin ("specs.txt");\n’)
f.write("     if (fin.is_open())\n\
     {\n\
       while (! fin.eof())\n\
       {\n\
         getline (fin,buffer);\n\
 if (buffer[0] !=’#’)\n\
   {\n\
   specs[count] = buffer;\n\
   count++;\n\
   }\n\
       }\n\
       fin.close();\n\
     }\n\
     \n\
     else cout << \"Unable to open file\";\n")

# Write Time to file
#  Time is a misnomer - this is a measure of the the number
#  of time steps used in the problem
f.write(‘    Time = atoi(specs[0].c_str());\n’)
# Write skip to file
#  Skip is a dummy variable to allow the use of various parts
#  of a given data file
f.write(‘    skip = atoi(specs[1].c_str());\n’)
# Write hstep to file
#  Hstep is the time-step of the discretization
f.write(‘    hstep = atof(specs[2].c_str());\n\n’)

# Write open data file to file
f.write(‘    string filename;\n’)
f.write(‘    int ret;\n’)

# Data for each variable that is being coupled in to the vector field
for i in range(nU):
    f.write(‘\
```

```
    %s = new double[2*Time+1];\n\
    %s = new double[skip];\n\n\
    FILE *pFile%d;\n' % (discretize.Ldata[i],discretize.Ldata[i]+
    'dummy',i))
    f.write('\
    filename = specs[%d];\n' % (3+i))
    f.write('\
    pFile%d = fopen(filename.c_str(),"r");\n' % i)

# Read data from data file
    temp1 = "%lf"
    f.write('\n\
    for(Index jt=0;jt<skip;jt++)\n\
{\n\
ret = fscanf (pFile%d, "%s", &%s[jt]);\n\
if (ret == EOF) break;\n\
}\n\
    for(Index jt=0;jt<2*Time+1;jt++)\n\
{\n\
ret = fscanf (pFile%d, "%s", &%s[jt]);\n\
if (ret == EOF) break;\n\
}\n\
    fclose (pFile%d);\n' % (i,temp1, discretize.Ldata[i]+
    'dummy',i,
    temp1,discretize.Ldata[i],i))
#########  End for loop #############

# Open data file for stimulus
for i in range(nI):
    f.write('\
    %s = new double[2*Time+1];\n\
    %s = new double[skip];\n\n\
    FILE *qFile%d;\n' % (discretize.Lstimuli[i],
    discretize.Lstimuli[i]+'dummy',i))
    f.write('\
    filename = specs[%d];\n' % (3+nU+i))
    f.write('\
    qFile%d = fopen(filename.c_str(),"r");\n' % i)
# Read stimuli data
    temp1 = "%lf"
    f.write('\n\
    for(Index jt=0;jt<skip;jt++)\n\
        {\n\
```

```
ret = fscanf (qFile%d, "%s", &%s[jt]);\n\
if (ret == EOF) break;\n\
        }\n\
    for(Index jt=0;jt<2*Time+1;jt++)\n\
        {\n\
ret = fscanf (qFile%d, "%s", &%s[jt]);\n\
if (ret == EOF) break;\n\
}\n\
    fclose (qFile%d);\n' % (i,temp1,discretize.Lstimuli[i]+'dummy',
    i,temp1,discretize.Lstimuli[i],i))
#########  End for loop #############

# Read in the initial and boundary conditions for all variables
#  into arrays
f.write('\
    int rows = nY+2*nU+nP;\n\
    bounds = new double*[rows];\n\
    for (Index i=0;i<rows;i++) bounds[i] = new double[4];\n\
    int toggle=0;\n\
    if (specs[3+nU+nI] == "1") toggle = 1;\n\
    int counter;\n\
    for(Index k=0;k<rows;k++)\n\
       {\n\
       counter=0;\n\
       char* tmp = new char[specs[4+nU+nI+toggle+k].size()+1];\n\
       strcpy( tmp, specs[4+nU+nI+toggle+k].c_str() );\n\
       char *ptr = strtok(tmp,",");\n\
       while(ptr != 0) {\n\
          if(counter<4) {\n\
     bounds[k][counter] = atof(ptr);\n\
     }\n\
  ptr = strtok(0,",");\n\
  counter++;\n\
          }\n\
    }\n\n')
# If initial conditions are in a data file, read in the data file
f.write('\
    if (specs[3+nU+nI] == "1")\n\
       {\n\
       filename = specs[4+nU+nI];\n\
       }\n\n')
```

```
f.write(‘\
}\n\n’)

f.write(‘// destructor\n\
%s_NLP::~%s_NLP()\n\
{\n\
  delete [] K11val;\n\
  delete [] K11val2;\n\
  delete [] K11valp1;\n\
  delete [] dK11val;\n\
  delete [] dK11val2;\n\
  delete [] dK11valp1;\n\
  delete [] Xdval;\n\
  delete [] Xdval2;\n\
  delete [] Xdvalp1;\n\
  delete [] Xval;\n\
  delete [] Xval2;\n\
  delete [] Xvalp1;\n\
  delete [] Pval;\n\
  delete [] Ival;\n\
  delete [] Ival2;\n\
  delete [] Ivalp1;\n\
  delete [] specs;\n’ % (probu, probu))
for i in range(nU):
    f.write(‘\
    delete [] %s;\n\
    delete [] %s;\n’ % (discretize.Ldata[i],discretize.Ldata[i]+
    ’dummy’))
for i in range(nI):
    f.write(‘\
    delete [] %s;\n\
    delete [] %s;\n’ % (discretize.Lstimuli[i],discretize.Lstimuli[i]+
    ’dummy’))
f.write(‘\
  int rows = nY+2*nU+nP;\n\
  for (Index i=0;i<rows;i++) delete [] bounds[i];\n\
  delete [] bounds;\n’)
f.write(‘\n\
}\n\n’)
# Start to write individual functions

# GET_NLP_INFO
```

```
f.write(‘// returns the size of the problem\n\
bool %s_NLP::get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,\n\
Index& nnz_h_lag, IndexStyleEnum& index_style)\n\n’ % probu)
# Number of variables
alpha = 2*(nY+2*nU)
beta = nY+2*nU+nP
f.write(‘{\n\
  // Number of variables\n\
  n = %d*Time+%d;\n’ % (alpha,beta))

# Number of equality constraints
gamma = 2*nY+nU
f.write(‘\n\
  // Number of equality constraints\n\
  m = %d*Time;\n’ % gamma)

# Number of Jacobian nonzero entries
theta = len(discretize.VJac)
f.write(‘\n\
  // Number of Jacobian nonzero entries\n\
  nnz_jac_g = %d*Time;\n’ % theta)

# Number of Hessian non-zeros in lower left of diagonal
omega = discretize.index
zeta = discretize.oddball

# Index is the total number of non-zero elements for a
# given time-step, and oddball tracks which of these is a
# parameter/parameter derivative and thus does not get
# spread over multiple time steps.

f.write(‘\n\
  // Number of Hessian nonzero entries\n\
  nnz_h_lag = %d*Time+%d;\n’ % ((omega-zeta),(omega-zeta)/2+zeta))

f.write(‘\n\
  // use the C style indexing (0-based)\n\
  index_style = TNLP::C_STYLE;\n\n\
  return true;\n\
}\n\n\n’)
```

```
# GET_BOUNDS_INFO


f.write(‘// returns the variable bounds\n\
bool %s_NLP::get_bounds_info(Index n, Number* x_l, Number* x_u,\n\
                             Index m, Number* g_l, Number* g_u)\n\
{\n\
  // Here, the n and m we gave IPOPT in get_nlp_info are passed back
  to us.\n\
  // If desired, we could assert to make sure they are what we think
  they are.\n’ % probu)
f.write(‘  assert(n == %d*Time+%d);\n’ % (alpha, beta))
f.write(‘  assert(m == %d*Time);\n\n’ % gamma)

# This takes the bounds information read in from specs.txt, and puts
#  it into the correct spot in a bounds array for all time points.

f.write(‘  for(Index jt=0;jt<Time+1;jt++) {\n’)
f.write(‘     for(Index var=0;var<nY;var++) {\n’)
f.write(‘        // Bounds for x\n’)
f.write(‘        x_l[(Time+1)*var+jt]=bounds[var][0];\n’)
f.write(‘        x_u[(Time+1)*var+jt]=bounds[var][1];\n’)
f.write(‘        // Bounds for midpoints\n’)
f.write(‘        if(jt<Time) {\n\
       x_l[(Time+1)*(nY+2*nU)+Time*var+jt]=bounds[var][0];\n\
       x_u[(Time+1)*(nY+2*nU)+Time*var+jt]=bounds[var][1];\n\
       }\n\
    }\n’)

f.write(‘     for(Index con=0;con<2*nU;con++) {\n’)
f.write(‘       // Bounds for k\n’)
f.write(‘       x_l[(Time+1)*(nY+con)+jt]=bounds[nY+con][0];\n’)
f.write(‘       x_u[(Time+1)*(nY+con)+jt]=bounds[nY+con][1];\n’)
f.write(‘       // Bounds for midpoints\n’)
f.write(‘       if(jt<Time) {\n\
          x_l[(Time+1)*(nY+2*nU)+Time*(nY+con)+jt]=bounds[nY+
          con][0];\n\
  x_u[(Time+1)*(nY+2*nU)+Time*(nY+con)+jt]=bounds[nY+con][1];\n\
  }\n\
     }\n\n’)
f.write(‘  } // End for loop\n\n’)
f.write(‘     for(Index par=0;par<nP;par++) {\n’)
```

```
f.write(‘        // Bounds for parameters\n’)
f.write(‘        x_l[2*Time*(nY+2*nU)+nY+2*nU+par]=bounds[nY+2*nU+
par][0];\n’)
f.write(‘        x_u[2*Time*(nY+2*nU)+nY+2*nU+par]=bounds[nY+2*nU+
par][1];\n\
             }\n\n’)

f.write(‘  // All constraints are equality constraints, so we set \n\
  // the upper and lower bound to the same value\n\
  // For noisy problems, allow these to be inequality constraints\n\
  //  as specified in the specs.txt file\n\
  for(Index jt=0; jt<Time; jt++) {\n\
     for(Index nn=0; nn<nY; nn++) {\n\
        g_l[%d*jt+nn] = g_l[%d*jt+nn+nY] = -bounds[nn][3];\n\
        g_u[%d*jt+nn] = g_u[%d*jt+nn+nY] =  bounds[nn][3];\n\
        }\n\
  // Add in constraints for the control midpoint\n\
     for(Index nn=0; nn<nU; nn++) {\n\
        g_l[%d*jt+2*nY+nn] = -bounds[nY+nn][3];\n\
        g_u[%d*jt+2*nY+nn] =  bounds[nY+nn][3];\n\
        }\n\
     }\n\
  return true;\n\
}\n\n\n’ % (2*nY+nU, 2*nY+nU, 2*nY+nU, 2*nY+nU, 2*nY+nU, 2*nY+nU))


# GET_STARTING_POINT

f.write(‘// returns the initial point for the problem\n\
bool %s_NLP::get_starting_point(Index n, bool init_x, Number* x,\n\
                         bool init_z, Number* z_L, Number* z_U,\n\
                         Index m, bool init_lambda,\n\
                         Number* lambda)\n\
{\n\
  assert(init_x == true);\n\
  assert(init_z == false);\n\
  assert(init_lambda == false);\n\n\
  for (Index i=0; i<n; i++) {\n\
        x[i] = 0.0;\n\
      }\n\n’ % probu)
f.write(‘\
      int ROWS = 2*Time+1;\n\
      int COLS = nY;\n\
```

```
    \n\
      double **init = new double* [ROWS];\n\
      for(Index i=0;i<ROWS;i++) init[i] = new double[COLS];\n\
      \n\
      string filename;\n\
      filename = specs[4+nU+nI];')

# To start from an initial guess given in a separate data file:
#  Read in the data file
temp1 = "%lf"
f.write('\
      if (specs[3+nU+nI] =="1")\n\
      {\n\
      FILE *initFILE;\n\
      int ret;\n\
      initFILE = fopen(filename.c_str(),"r");\n\
    \n\
      for(Index jt=0;jt<2*Time+1;jt++)\n\
          {\n\
  ret = fscanf (initFILE,"')
for i in range(nY):
   f.write('%s ' % temp1)
f.write('"')
for i in range(nY):
   f.write(',&init[jt][%d]' % i)
f.write(');\n\
if (ret == EOF) break;\n\
        }\n\
    fclose (initFILE);\n\
    }\n\n')

# Set the initial starting point into the x[] array, either from
#  the numbers given in specs.txt or from an initial data file

f.write('  for(Index jt=0;jt<Time+1;jt++) {\n')
f.write('     for(Index var=0;var<nY;var++) {\n')
f.write('       // Initial conditions for x\n')
f.write('       if (specs[3+nU+nI] == "1")\n\
                  {\n\
  x[(Time+1)*var+jt] = init[2*jt][var];\n\
  }\n\
        else\n\
          {\n\
```

```
  x[(Time+1)*var+jt] = bounds[var][2];\n\
  }\n')
f.write('       // Initial conditions for midpoints\n')
f.write('       if(jt<Time) {\n\
                  if (specs[3+nU+nI] == "1")\n\
    {\n\
    x[(Time+1)*(nY+2*nU)+Time*var+jt] = init[2*jt+1][var];\n\
    }\n\
  else\n\
    {\n\
    x[(Time+1)*(nY+2*nU)+Time*var+jt] = bounds[var][2];\n\
    }\n\
  }\n\
}\n')

f.write('     for(Index cup=0;cup<2*nU;cup++) {\n')
f.write('       // Initial conditions for k\n')
f.write('       x[(Time+1)*(cup+nY)+jt]=bounds[cup+nY][2];\n')
f.write('       // Initial conditions for midpoints\n')
f.write('       if(jt<Time) {\n\
                  x[(Time+1)*(nY+2*nU)+Time*(cup+nY)+jt]=
                  bounds[cup+nY][2];\n\
          }\n\
      }\n')

f.write('  } // End for loop\n\n')

f.write('     for(Index par=0;par<nP;par++) {\n')
f.write('     // Initial conditions for p%d\n' % (i+1))
f.write('     x[2*Time*(nY+2*nU)+nY+2*nU+par]=
bounds[nY+2*nU+par][2];\n\
              }\n\n')
f.write('  for(Index i=0;i<ROWS;i++) delete [] init[i];\n\
   delete [] init;\n')
f.write('  return true;\n\
}\n\n\n')



# EVAL_F
# Subroutine to calculate the objective value
# Here, and in the following subroutines, strings from the result
#  of symbolic discretization and differentiation in discretize.py
```

```
#  are inserted to the code.

f.write(‘// returns the value of the objective function\n\
bool %s_NLP::eval_f(Index n, const Number* x, bool new_x,
Number& obj_value)\n\
{\n’ % probu)
f.write(‘  assert(n == %d*Time+%d);\n’ % (alpha, beta))
f.write(‘  obj_value = 0;\n\n’)
f.write(‘  for(Index jt=0;jt<Time;jt++) {\n\n’)
f.write(‘     for(Index i=0;i<nY;i++) {\n’)
f.write(‘        Xval[i] = x[jt + i*(Time+1)];\n’)
f.write(‘        Xvalp1[i] = x[jt + i*(Time+1) + 1];\n’)
f.write(‘        Xval2[i] = x[(Time+1)*(nY+2*nU) + jt + i*(Time)];\n’)
f.write(‘     } //end for loop\n\n’)

f.write(‘\n’)
f.write(‘  for(Index i=0;i<nU;i++) {\n’)
f.write(‘    K11val[i]=x[jt+nY*(Time+1)+2*i*(Time+1)];\n’)
f.write(‘    K11valp1[i]=x[jt+nY*(Time+1)+2*i*(Time+1)+1];\n’)
f.write(‘    K11val2[i]=x[(Time+1)*(nY+2*nU)+(nY+2*i)*Time+jt];\n’)
f.write(‘    dK11val[i]=x[jt+(nY+2*i+1)*(Time+1)];\n’)
f.write(‘    dK11valp1[i]=x[jt+(nY+2*i+1)*(Time+1)+1];\n’)
f.write(‘    dK11val2[i]=x[(Time+1)*(nY+2*nU)+(nY+2*i+1)*Time+jt];\n’)

f.write(‘     } //end for loop\n\n’)

for i in range(nU):
    f.write(‘     Xdval[%d]=%s[2*jt];\n’%(i,discretize.Ldata[i]))
    f.write(‘     Xdval2[%d]=%s[2*jt+1];\n’%(i,discretize.Ldata[i]))
    f.write(‘     Xdvalp1[%d]=%s[2*jt+2];\n’%(i,discretize.Ldata[i]))

for i in range(nI):
    f.write(‘     Ival[%d]=%s[2*jt];\n’%(i,discretize.Lstimuli[i]))
    f.write(‘     Ival2[%d]=%s[2*jt+1];\n’%(i,discretize.Lstimuli[i]))
    f.write(‘     Ivalp1[%d]=%s[2*jt+2];\n’%(i,discretize.Lstimuli[i]))

f.write(‘\n’)
f.write(‘     for(Index i=0;i<nP;i++) {\n’)
f.write(‘        Pval[i] = x[(2*Time+1)*(nY+2*nU)+i];\n’)
f.write(‘     } //end for loop\n\n’)
f.write(‘\n’)
f.write(‘    obj_value += %s + %s;\n\n’ % (discretize.strObj[0][0],
discretize.strObj[0][2]))
```

```
f.write('  } //end for loop\n\n')

# Add code for last element

f.write('// Add last element\n')
f.write('     for(Index i=0;i<nY;i++) {\n')
f.write('        Xval[i] = x[Time + i*(Time+1)];\n')
f.write('        Xvalp1[i] = 0;\n')
f.write('        Xval2[i] = 0;\n')
f.write('     } //end for loop\n\n')

f.write('\n')
f.write('   for(Index i=0;i<nU;i++) {\n')
f.write('     K11val[i] = x[Time+nY*(Time+1)+2*i*(Time+1)];\n')
f.write('     K11valp1[i] = 0;\n')
f.write('     K11val2[i] = 0;\n')
f.write('     dK11val[i] = x[Time + (nY+2*i+1)*(Time+1)];\n')
f.write('     dK11valp1[i] = 0;\n')
f.write('     dK11val2[i] = 0;\n')

f.write('     } //end for loop\n\n')

for i in range(nU):
    f.write('     Xdval[%d]=%s[2*Time];\n'%(i,discretize.Ldata[i]))
    f.write('     Xdval2[%d] = 0;\n' % i)
    f.write('     Xdvalp1[%d] = 0;\n' % i)

for i in range(nI):
    f.write('     Ival[%d]=%s[2*Time];\n'%(i,discretize.Lstimuli[i]))
    f.write('     Ival2[%d] = 0;\n' % i)
    f.write('     Ivalp1[%d] = 0;\n' % i)

f.write('\n')
f.write('     for(Index i=0;i<nP;i++) {\n')
f.write('        Pval[i] = x[(2*Time+1)*(nY+2*nU)+i];\n')
f.write('     } //end for loop\n\n')
f.write('\n')

f.write('  obj_value += %s + %s;\n\n' % (discretize.strObj[0][0],
discretize.strObj[0][2]))

# Adding a line to divide the overall objective function by 2T + 1
```

```
# This normalizes the objective function

f.write('  obj_value = obj_value/(2*Time+1);\n\n')

f.write('  return true;\n\
}\n\n\n')



# EVAL_GRAD_F

f.write('// return the gradient of the objective function grad_{x}
f(x)\n\
bool %s_NLP::eval_grad_f(Index n, const Number* x, bool new_x,
Number* grad_f)\n\
{\n' % probu)
f.write('  assert(n == %d*Time+%d);\n\n' % (alpha, beta))

f.write('  for(Index i=0;i<n;i++) {\n')
f.write('     grad_f[i] = 0;\n')
f.write('  }\n\n')

f.write('  for(Index jt=0;jt<Time;jt++) {\n\n')
f.write('     for(Index i=0;i<nY;i++) {\n')
f.write('        Xval[i] = x[jt + i*(Time+1)];\n')
f.write('        Xvalp1[i] = x[jt + i*(Time+1) + 1];\n')
f.write('        Xval2[i] = x[(Time+1)*(nY+2*nU)+jt+i*(Time)];\n')
f.write('     } //end for loop\n\n')

f.write(' for(Index i=0;i<nU;i++) {\n')
f.write('  K11val[i]=x[jt+nY*(Time+1)+2*i*(Time+1)];\n')
f.write('  K11valp1[i]=x[jt+nY*(Time+1)+2*i*(Time+1)+1];\n')
f.write('  K11val2[i]=x[(Time+1)*(nY+2*nU)+(nY+2*i)*Time+jt];\n')
f.write('  dK11val[i]=x[jt + (nY+2*i+1)*(Time+1)];\n')
f.write('  dK11valp1[i]=x[jt + (nY+2*i+1)*(Time+1)+1];\n')
f.write('  dK11val2[i]=x[(Time+1)*(nY+2*nU)+(nY+2*i+1)*Time+jt];\n')

f.write('     } //end for loop\n\n')

for i in range(nU):
    f.write('   Xdval[%d]=%s[2*jt];\n'%(i,discretize.Ldata[i]))
    f.write('   Xdval2[%d]=%s[2*jt+1];\n'%(i,discretize.Ldata[i]))
    f.write('   Xdvalp1[%d]=%s[2*jt+2];\n'%(i,discretize.Ldata[i]))
```

```
for i in range(nI):
    f.write(‘  Ival[%d]=%s[2*jt];\n’%(i,discretize.Lstimuli[i]))
    f.write(‘  Ival2[%d]=%s[2*jt+1];\n’%(i,discretize.Lstimuli[i]))
    f.write(‘  Ivalp1[%d]=%s[2*jt+2];\n’%(i,discretize.Lstimuli[i]))

f.write(‘\n’)
f.write(‘     for(Index i=0;i<nP;i++) {\n’)
f.write(‘        Pval[i] = x[(2*Time+1)*(nY+2*nU)+i];\n’)
f.write(‘     } //end for loop\n’)
f.write(‘\n’)

VObj = discretize.VObj

for i in range(len(VObj)):
    if VObj[i][1] < nY+2*nU:
if VObj[i][2] == 0:
    f.write(‘    grad_f[jt+%d*(Time+1)] = (%s)/(2*Time+1);\n’ %
    (VObj[i][1], VObj[i][0]))
        elif VObj[i][2] == 2:
    f.write(‘    grad_f[(Time+1)*(2*nU+nY) + %d*Time + jt] =
    (%s)/(2*Time+1);\n’ % (VObj[i][1], VObj[i][0]))
    else:
f.write(‘     grad_f[(2*Time+1)*(nY+2*nU)+%d] = (%s)/(2*Time+1);
\n’ % (VObj[i][1]-nY-nU, VObj[i][0]))
f.write(‘\n’)
f.write(‘  } //end for loop\n\n’)

# Add code for last gradient element

f.write(‘// Add last element\n’)
f.write(‘     for(Index i=0;i<nY;i++) {\n’)
f.write(‘        Xval[i] = x[Time + i*(Time+1)];\n’)
f.write(‘        Xvalp1[i] = 0;\n’)
f.write(‘        Xval2[i] = 0;\n’)
f.write(‘     } //end for loop\n\n’)

f.write(‘     for(Index i=0;i<nU;i++) {\n’)
f.write(‘        K11val[i] = x[Time+nY*(Time+1)+2*i*(Time+1)];\n’)
f.write(‘        K11valp1[i] = 0;\n’)
f.write(‘        K11val2[i] = 0;\n’)
f.write(‘        dK11val[i] = x[Time + (nY+2*i+1)*(Time+1)];\n’)
f.write(‘        dK11valp1[i] = 0;\n’)
```

```
f.write(‘        dK11val2[i] = 0;\n’)

f.write(‘     } //end for loop\n\n’)

for i in range(nU):
    f.write(‘   Xdval[%d] = %s[2*Time];\n’%(i,discretize.Ldata[i]))
    f.write(‘   Xdval2[%d] = 0;\n’ % i)
    f.write(‘   Xdvalp1[%d] = 0;\n’ % i)

for i in range(nI):
    f.write(‘   Ival[%d]=%s[2*Time];\n’%(i,discretize.Lstimuli[i]))
    f.write(‘   Ival2[%d]=0;\n’ % i)
    f.write(‘   Ivalp1[%d]=0;\n’ % i)

f.write(‘\n’)
f.write(‘     for(Index i=0;i<nP;i++) {\n’)
f.write(‘        Pval[i] = x[(2*Time+1)*(nY+2*nU)+i];\n’)
f.write(‘     } //end for loop\n\n’)
f.write(‘\n’)

for i in range(len(VObj)):
    if VObj[i][1] < nY+2*nU:
if VObj[i][2] == 0:
    f.write(‘    grad_f[Time+%d*(Time+1)] = (%s)/(2*Time+1);\n’ %
    (VObj[i][1], VObj[i][0]))

f.write(‘\n’)
f.write(‘  return true;\n\
}\n\n\n’)



# EVAL_G

f.write(‘// return the value of the constraints: g(x)\n\
bool %s_NLP::eval_g(Index n, const Number* x, bool new_x, Index m,
Number* g)\n\
{\n’ % probu)
f.write(‘  assert(n == %d*Time+%d);\n’ % (alpha, beta))
f.write(‘  assert(m == %d*Time);\n\n’ % gamma)

# Put in constraint functions
```

```
f.write(‘  for(Index jt=0;jt<Time;jt++) {\n\n’)
f.write(‘     for(Index i=0;i<nY;i++) {\n’)
f.write(‘        Xval[i] = x[jt + i*(Time+1)];\n’)
f.write(‘        Xvalp1[i] = x[jt + i*(Time+1) + 1];\n’)
f.write(‘        Xval2[i]=x[(Time+1)*(nY+2*nU)+jt+i*(Time)];\n’)
f.write(‘     } //end for loop\n\n’)

f.write(‘ for(Index i=0;i<nU;i++) {\n’)
f.write(‘   K11val[i]=x[jt+nY*(Time+1)+2*i*(Time+1)];\n’)
f.write(‘   K11valp1[i]=x[jt+nY*(Time+1)+2*i*(Time+1)+1];\n’)
f.write(‘   K11val2[i]=x[(Time+1)*(nY+2*nU)+(nY+2*i)*Time+jt];\n’)
f.write(‘   dK11val[i]=x[jt+(nY+2*i+1)*(Time+1)];\n’)
f.write(‘   dK11valp1[i]=x[jt+(nY+2*i+1)*(Time+1)+1];\n’)
f.write(‘   dK11val2[i]=x[(Time+1)*(nY+2*nU)+(nY+2*i+1)*Time+jt];\n’)

f.write(‘     } //end for loop\n\n’)

for i in range(nU):
    f.write(‘     Xdval[%d] = %s[2*jt];\n’ %
    (i,discretize.Ldata[i]))
    f.write(‘     Xdval2[%d] = %s[2*jt+1];\n’ %
     (i,discretize.Ldata[i]))
    f.write(‘     Xdvalp1[%d] = %s[2*jt+2];\n’ %
    (i,discretize.Ldata[i]))

for i in range(nI):
    f.write(‘   Ival[%d]=%s[2*jt];\n’%(i,discretize.Lstimuli[i]))
    f.write(‘   Ival2[%d]=%s[2*jt+1];\n’%(i,discretize.Lstimuli[i]))
    f.write(‘   Ivalp1[%d]=%s[2*jt+2];\n’%(i,discretize.Lstimuli[i]))

f.write(‘\n’)
f.write(‘     for(Index i=0;i<nP;i++) {\n’)
f.write(‘        Pval[i] = x[(2*Time+1)*(nY+2*nU)+i];\n’)
f.write(‘     } //end for loop\n’)
f.write(‘\n’)

AllCon = discretize.strAllCon

for i in range(len(AllCon)):
    f.write(‘     g[%d*jt+%d] = %s + %s + %s;\n’ % (len(AllCon), i,
     AllCon[i][0], AllCon[i][1], AllCon[i][2]))

f.write(‘\n’)
```

```
f.write('  } //end for loop\n\n')


f.write('  return true;\n\
}\n\n\n')



# EVAL_JAC_G

f.write('// return the structure or values of the jacobian\n\
bool %s_NLP::eval_jac_g(Index n, const Number* x, bool new_x,\n\
             Index m, Index nele_jac, Index* iRow, Index* jCol,\n\
             Number* values)\n\
{\n\n\
if (values == NULL) {\n\
   // return the structure of the jacobian\n\
   for(Index jt=0;jt<Time;jt++) {\n' % probu)
# Jacobian index structure

for i in range(len(discretize.VJac)):
   f.write('      iRow[%d*jt+%d] = %d+%d*jt;\n'\
            % (len(discretize.VJac),i,discretize.VJac[i][1],
            len(discretize.strAllCon)))
   f.write('      jCol[%d*jt+%d] = ' % (len(discretize.VJac),i))

   if discretize.VJac[i][2] < len(discretize.Lvars)+
   len(discretize.Lcouple):
       if discretize.VJac[i][3] == 0:
           f.write('(Time+1)*%d+jt;\n' % discretize.VJac[i][2])
       elif discretize.VJac[i][3] == 1:
           f.write('(Time+1)*%d+jt+1;\n' % discretize.VJac[i][2])
       elif discretize.VJac[i][3] == 2:
           f.write('(Time+1)*%d+Time*%d+jt;\n'\
                   % (len(discretize.Lvars)+len(discretize.Lcouple),
                    discretize.VJac[i][2]))
       else:
           f.write('Error %d\n' % i)
   else:
       f.write('2*Time*%d+%d;\n' % (len(discretize.Lvars)+
       len(discretize.Lcouple), discretize.VJac[i][2]))

f.write('      } // end for loop\n\n\
```

```
   } // end if\n\n\
else {\n\
   // return the values of the jacobian\n\
   for(Index jt=0;jt<Time;jt++) {\n')

# Jacobian values

f.write('     for(Index i=0;i<nY;i++) {\n')
f.write('        Xval[i] = x[jt + i*(Time+1)];\n')
f.write('        Xvalp1[i] = x[jt + i*(Time+1) + 1];\n')
f.write('        Xval2[i] = x[(Time+1)*(nY+2*nU) + jt + i*(Time)];\n')
f.write('     } //end for loop\n\n')

f.write(' for(Index i=0;i<nU;i++) {\n')
f.write('   K11val[i]=x[jt+nY*(Time+1)+2*i*(Time+1)];\n')
f.write('   K11valp1[i]=x[jt+nY*(Time+1)+2*i*(Time+1)+1];\n')
f.write('   K11val2[i]=x[(Time+1)*(nY+2*nU)+(nY+2*i)*Time+jt];\n')
f.write('   dK11val[i]=x[jt+(nY+2*i+1)*(Time+1)];\n')
f.write('   dK11valp1[i]=x[jt+(nY+2*i+1)*(Time+1)+1];\n')
f.write('   dK11val2[i]=x[(Time+1)*(nY+2*nU)+(nY+2*i+1)*Time+jt];\n')

f.write('     } //end for loop\n\n')

for i in range(nU):
    f.write('     Xdval[%d] = %s[2*jt];\n' %
    (i,discretize.Ldata[i]))
    f.write('     Xdval2[%d] = %s[2*jt+1];\n' %
     (i,discretize.Ldata[i]))
    f.write('     Xdvalp1[%d] = %s[2*jt+2];\n' %
    (i,discretize.Ldata[i]))

for i in range(nI):
    f.write('   Ival[%d]=%s[2*jt];\n'%(i,discretize.Lstimuli[i]))
    f.write('   Ival2[%d]=%s[2*jt+1];\n'%(i,discretize.Lstimuli[i]))
    f.write('   Ivalp1[%d]=%s[2*jt+2];\n'%(i,discretize.Lstimuli[i]))

f.write('\n')
f.write('     for(Index i=0;i<nP;i++) {\n')
f.write('        Pval[i] = x[(2*Time+1)*(nY+2*nU)+i];\n')
f.write('     } //end for loop\n')
f.write('\n')

VJac = discretize.VJac
```

```
for i in range(len(VJac)):
    f.write(‘     values[%d*jt+%d] = ’ % (len(VJac),i))
    f.write(‘%s;\n’ % VJac[i][0])

f.write(‘\n’)
f.write(‘  } //end for loop\n\n’)
f.write(‘  } //end else\n\n’)


f.write(‘  return true;\n\
}\n\n\n’)



# EVAL_H

f.write(‘// return the structure or values of the hessian\n\
bool %s_NLP::eval_h(Index n, const Number* x, bool new_x,\n\
             Number obj_factor, Index m, const Number* lambda,\n\
             bool new_lambda, Index nele_hess, Index* iRow,\n\
             Index* jCol, Number* values)\n\
{\n\n\
if (values == NULL) {\n\
   // return the structure.  This is a symmetric matrix,
    fill in the lower left\n\
   // triangle only.\n\n’ % probu)

# Set up dictionaries to mark start point and length of each
# Hessian entry dictlength will be a string of either 1, Time,
# or Time +1 depending on how many entries in the Hessian
# there are for an element dictstart determines where each
# of the parts start based on what has come before

f.write(‘//Each non-zero element has its own explicit loop\n\
   // since each element needs a different number of matrix
   elements\n\n’)

dictstart = {}
dictlength = {}

VHes = discretize.VHes
```

```
# sma, med, lar track how many 1, T, T+1 there are
sma = 0
med = 0
lar = 0

for i in range(len(VHes)):
    if VHes[i][6] == 1:
# VHes[i][6] denotes whether this is the first element
row = VHes[i][2]
col = VHes[i][3]
mid = VHes[i][4]
count = VHes[i][0]
start = '%d*(Time+1)+%d*(Time)+%d' % (lar,med,sma)
if mid == -1:
    length = '1'
    sma = sma + 1
if mid == 2:
    length = 'Time'
    med = med + 1
if mid == 0:
    length = 'Time+1'
    lar = lar + 1
        d1 = {count:start}
        d2 = {count:length}
dictstart.update(d1)
dictlength.update(d2)
f.write(`\n   for(Index jt=0;jt<%s;jt++) {\n' % length)
f.write(`     iRow[%s+jt] = ' % start)
        if row < nY+2*nU:
    if mid == 0:
f.write(`(Time+1)*%d+jt;\n' % row)
    elif mid == 2:
f.write(`(Time+1)*%d+Time*%d+jt;\n' % (nY+2*nU,row))
else:
    f.write(`2*Time*%d+%d;\n' % (nY+2*nU, row))
f.write(`     jCol[%s+jt] = ' % start)
        if col < nY+2*nU:
    if mid == 0:
f.write(`(Time+1)*%d+jt;\n' % col)
    elif mid == 2:
f.write(`(Time+1)*%d+Time*%d+jt;\n' % (nY+2*nU,col))
else:
    f.write(`2*Time*%d+%d;\n' % (nY+2*nU, col))
```

```
f.write(‘   }\n’)

f.write(‘}\n\n\
else {\n\
  // return the values.
  This is a symmetric matrix, fill the lower left\n\
  // triangle only\n\
  // initialize the values array\n\
  // Point to the initial starting spot for the Hessian elements\n\n\
  for(Index jt=0;jt<%d*Time+%d;jt++) values[jt] = 0.;
   // Initialize matrix’ % ((omega-zeta),(omega-zeta)/2+zeta))
f.write(‘\n\n   // fill the objective portion\n\n’)
Objrow = 2*nY+nU
# Doing the singletons first - should not be many
for i in range(len(VHes)):
    if VHes[i][1] == Objrow:
mid = VHes[i][4]
count = VHes[i][0]
string = VHes[i][5]
start = dictstart[count]
if mid == -1:
    f.write(‘   values[%s] += ’ % start)
    f.write(‘obj_factor*(%s)/(2*Time+1);\n\n’ % string)
# Now loop all other entries over Time

f.write(‘  for(Index jt=0;jt<Time;jt++) {\n\n’)
f.write(‘    for(Index i=0;i<nY;i++) {\n’)
f.write(‘      Xval[i]=x[jt + i*(Time+1)];\n’)
f.write(‘      Xvalp1[i]=x[jt + i*(Time+1) + 1];\n’)
f.write(‘      Xval2[i]=x[(Time+1)*(nY+2*nU) + jt + i*(Time)];\n’)
f.write(‘     } //end for loop\n\n’)

f.write(‘ for(Index i=0;i<nU;i++) {\n’)
f.write(‘  K11val[i]=x[jt+nY*(Time+1)+2*i*(Time+1)];\n’)
f.write(‘  K11valp1[i]=x[jt+nY*(Time+1)+2*i*(Time+1)+1];\n’)
f.write(‘  K11val2[i]=x[(Time+1)*(nY+2*nU)+(nY+2*i)*Time+jt];\n’)
f.write(‘  dK11val[i]=x[jt+(nY+2*i+1)*(Time+1)];\n’)
f.write(‘  dK11valp1[i]=x[jt + (nY+2*i+1)*(Time+1)+1];\n’)
f.write(‘  dK11val2[i]=x[(Time+1)*(nY+2*nU)+(nY+2*i+1)*Time+jt];\n’)

f.write(‘     } //end for loop\n\n’)
```

```
for i in range(nU):
    f.write(‘     Xdval[%d] = %s[2*jt];\n’ %
    (i,discretize.Ldata[i]))
    f.write(‘     Xdval2[%d] = %s[2*jt+1];\n’ %
     (i,discretize.Ldata[i]))
    f.write(‘     Xdvalp1[%d] = %s[2*jt+2];\n’ %
    (i,discretize.Ldata[i]))

for i in range(nI):
    f.write(‘  Ival[%d]=%s[2*jt];\n’%(i,discretize.Lstimuli[i]))
    f.write(‘  Ival2[%d]=%s[2*jt+1];\n’%(i,discretize.Lstimuli[i]))
    f.write(‘  Ivalp1[%d]=%s[2*jt+2];\n’%(i,discretize.Lstimuli[i]))
f.write(‘\n’)
f.write(‘     for(Index i=0;i<nP;i++) {\n’)
f.write(‘        Pval[i] = x[(2*Time+1)*(nY+2*nU)+i];\n’)
f.write(‘     } //end for loop\n’)
f.write(‘\n’)

for i in range(len(VHes)):
    if VHes[i][1] == Objrow:
mid = VHes[i][4]
count = VHes[i][0]
string = VHes[i][5]
start = dictstart[count]
if mid != -1:
    f.write(‘    values[%s+jt] += ’ % start)
            f.write(‘obj_factor*(%s)/(2*Time+1);\n’ % string)
f.write(‘   } //end loop over Time\n\n’)
f.write(‘   // Add elements for last time step\n\n’)

for i in range(len(VHes)):
    if VHes[i][1] == Objrow:
mid = VHes[i][4]
count = VHes[i][0]
string = VHes[i][5]
start = dictstart[count]
length = dictlength[count]
if mid == 0:
    f.write(‘    values[%s+%s-1] += ’ % (start, length))
    f.write(‘obj_factor*(%s)/(2*Time+1);\n’ % string)

# Now do the Hessian of the constraints
```

```
f.write(‘\n   // fill the constraint portions\n\n’)

# Loop over constraints

f.write(‘  for(Index jt=0;jt<Time;jt++) {\n\n’)
f.write(‘    for(Index i=0;i<nY;i++) {\n’)
f.write(‘      Xval[i]=x[jt + i*(Time+1)];\n’)
f.write(‘      Xvalp1[i]=x[jt + i*(Time+1) + 1];\n’)
f.write(‘      Xval2[i]=x[(Time+1)*(nY+2*nU) + jt + i*(Time)];\n’)
f.write(‘     } //end for loop\n\n’)

f.write(‘  for(Index i=0;i<nU;i++) {\n’)
f.write(‘   K11val[i]=x[jt+nY*(Time+1) + 2*i*(Time+1)];\n’)
f.write(‘   K11valp1[i]=x[jt+nY*(Time+1) + 2*i*(Time+1)+1];\n’)
f.write(‘   K11val2[i]=x[(Time+1)*(nY+2*nU) + (nY+2*i)*Time+jt];\n’)
f.write(‘   dK11val[i]=x[jt+(nY+2*i+1)*(Time+1)];\n’)
f.write(‘   dK11valp1[i]=x[jt+(nY+2*i+1)*(Time+1)+1];\n’)
f.write(‘   dK11val2[i]=x[(Time+1)*(nY+2*nU) (nY+2*i+1)*Time+jt];\n’)

f.write(‘     } //end for loop\n\n’)

for i in range(nU):
    f.write(‘     Xdval[%d] = %s[2*jt];\n’ %
    (i,discretize.Ldata[i]))
    f.write(‘     Xdval2[%d] = %s[2*jt+1];\n’ %
     (i,discretize.Ldata[i]))
    f.write(‘     Xdvalp1[%d] = %s[2*jt+2];\n’ %
    (i,discretize.Ldata[i]))

for i in range(nI):
    f.write(‘  Ival[%d]=%s[2*jt];\n’%(i,discretize.Lstimuli[i]))
    f.write(‘  Ival2[%d]=%s[2*jt+1];\n’%(i,discretize.Lstimuli[i]))
    f.write(‘  Ivalp1[%d]=%s[2*jt+2];\n’%(i,discretize.Lstimuli[i]))

f.write(‘\n’)
f.write(‘     for(Index i=0;i<nP;i++) {\n’)
f.write(‘        Pval[i] = x[(2*Time+1)*(nY+2*nU)+i];\n’)
f.write(‘     } //end for loop\n’)
f.write(‘\n’)

# Doing the singletons first - should not be many
for i in range(len(VHes)):
    if VHes[i][1] != Objrow:
```

```
mid = VHes[i][4]
count = VHes[i][0]
string = VHes[i][5]
constraint = VHes[i][1]
start = dictstart[count]
if mid == -1:
    f.write(‘   values[%s] += ’ % start)
    f.write(‘lambda[%d*jt+%d]*(%s);\n\n’ %
     (len(AllCon),constraint,string))

for i in range(len(VHes)):
    if VHes[i][1] != Objrow:
mid = VHes[i][4]
count = VHes[i][0]
string = VHes[i][5]
start = dictstart[count]
constraint = VHes[i][1]
if mid != -1:
    f.write(‘    values[%s+jt]   += ’ % start)
            f.write(‘lambda[%d*jt+%d]*(%s);\n’ % (len(AllCon),
            constraint, string))
    if mid == 0:
string = VHes[i][7]
f.write(‘    values[%s+jt+1] += ’ % start)
f.write(‘lambda[%d*jt+%d]*(%s);\n’ % (len(AllCon),
constraint, string))
f.write(‘   } // end for loop \n\n’)

f.write(‘  } // end else \n\n’)



f.write(‘   return true;\n\
}\n\n\n’)



# FINALIZE_SOLUTION


f.write(‘\
void %s_NLP::finalize_solution(SolverReturn status,\n\
  Index n, const Number* x, const Number* z_L, const Number* z_U,\n\
```

```
  Index m, const Number* g, const Number* lambda,\n\
  Number obj_value,\n\
  const IpoptData* ip_data,\n\
  IpoptCalculatedQuantities* ip_cq)\n\
{\n\
  // here is where the solution is written to file\n\n' % probu)


f.write('  FILE *OUTPUT1;\n')
f.write('  FILE *OUTPUT2;\n')
f.write('  FILE *OUTPUT3;\n')

temp1 = "%e"
temp2 = "%d"
temp3 = "\\n"

f.write('\n\
  OUTPUT1 = fopen ("param.dat","w");\n\
  OUTPUT2 = fopen ("data.dat","w");\n\
  OUTPUT3 = fopen ("Rvalue.dat","w");\n\n\
  // Final parameters\n\
  for (Index i=0;i<nP;i++) {\n\
    fprintf (OUTPUT1, "%s%s", x[(2*Time+1)*(nY+2*nU)+i]);\n\
    printf("Parameter[%s] = %s%s", i+1, x[(2*Time+1)*
    (nY+2*nU)+i]);\n\
    }\n\n' % (temp1,temp3,temp2,temp1,temp3))

f.write(' // Solution of the primal variables, x')
f.write('\n\n\
  for (Index i=0;i<Time;i++) {\n\
    fprintf(OUTPUT2,"%s ' % temp2)
for i in range(nY+2*nU):
  f.write('%s ' % temp1)
f.write('%s", 2*i, ' % temp3)
for i in range(nY):
  f.write('x[%d*(Time+1)+i], ' % i)
for i in range(nU):
  f.write('x[(nY+2*%d)*(Time+1)+i], ' % i)
for i in range(nU-1):
  f.write('%s[2*i], ' % discretize.Ldata[i])
f.write('%s[2*i]);\n' % discretize.Ldata[nU-1])

f.write('\n\
```

```
    fprintf(OUTPUT2,"%s ' % temp2)
for i in range(nY+2*nU):
  f.write('%s ' % temp1)
f.write('%s", 2*i+1, ' % temp3)
for i in range(nY):
  f.write('x[(nY+2*nU)*(Time+1)+%d*Time+i], ' % i)
for i in range(nU):
  f.write('x[(nY+2*nU)*(Time+1)+(nY+2*%d)*Time+i], ' % i)
for i in range(nU-1):
  f.write('%s[2*i+1], ' % discretize.Ldata[i])
f.write('%s[2*i+1]);\n' % discretize.Ldata[nU-1])


f.write('  }\n')

# Last time step

f.write('\n\
    fprintf(OUTPUT2,"%s ' % temp2)
for i in range(nY+2*nU):
  f.write('%s ' % temp1)
f.write('%s", 2*Time, ' % temp3)
for i in range(nY):
  f.write('x[%d*(Time+1)+Time], ' % i)
for i in range(nU):
  f.write('x[(nY+2*%d)*(Time+1)+Time], ' % i)
for i in range(nU-1):
  f.write('%s[2*Time], ' % discretize.Ldata[i])
f.write('%s[2*Time]);\n' % discretize.Ldata[nU-1])

f.write('\n\n')
f.write('  printf("%s%sObjective value%s");\n'%(temp3,temp3,temp3))
f.write('  printf("f(x*) = %s%s", obj_value);\n\n'%(temp1, temp3))


# Calculation of R

f.write(' // Calculation of synchronization error, R')

f.write('\n\n\
  for (Index jt=0;jt<Time;jt++) {\n\
     for(Index i=0;i<nY;i++) {\n\
        Xval[i] = x[jt + i*(Time+1)];\n\
```

```
}\n\
     \n\
     for(Index i=0;i<nU;i++) {\n\
        K11val[i] = x[jt + nY*(Time+1) + 2*i*(Time+1)];\n\
}\n\
     \n')

for i in range(nU):
    f.write('     Xdval[%d] = %s[2*jt];\n'%(i,discretize.Ldata[i]))

for i in range(nI):
    f.write('     Ival[%d] = %s[2*jt];\n'%(i,discretize.Lstimuli[i]))
f.write('\n')
f.write('     for(Index i=0;i<nP;i++) {\n')
f.write('        Pval[i] = x[(2*Time+1)*(nY+2*nU)+i];\n')
f.write('     } //end for loop\n')
f.write('\n')

strEqns = correlate.strEqns
cupEqns = correlate.cupEqns

for i in range(len(strEqns)):
    f.write('     Rvalue[%d] = pow(%s,2)/(pow(%s,2)+pow(%s,2));\n' %
     (i,strEqns[i],strEqns[i],cupEqns[i]))
f.write('\n')

f.write('  fprintf(OUTPUT3,"%s ' % temp2)
for i in range(nU):
  f.write('%s ' % temp1)
f.write('%s", 2*jt, ' % temp3)
for i in range(nU-1):
  f.write('Rvalue[%d], ' % i)
f.write('Rvalue[%d]);\n' % (nU-1))


f.write('  } //end jt for loop\n\n')


f.write('  fclose (OUTPUT1);\n')
f.write('  fclose (OUTPUT2);\n')
f.write('  fclose (OUTPUT3);\n')

f.write('}\n')
```

```
f.close( )
```

B.3 Makecpp.py

```
#########################################################
#
#  20 October 2009
#  Bryan A. Toth
#  University of California, San Diego
#  btoth@physics.ucsd.edu
#
#  This script writes the program file for a C++ IPOPT
#  program defined by the vector field in the file
#  equations.txt and written by the script makecode.py.
#  This file creates an instance of the non-linear file
#  defined by makehpp.py and makecode.py, and interfaces
#  with the IPOPT libraries necessary for solving the
#  optimization problem.
#
#  This script has been developed as part of a suite of
#  python scripts to define a dynamic parameter estimation
#  problem using the optimization software IPOPT, but is
#  generally applicable to any application needing
#  discretized derivatives of a vector field.
#
#########################################################

import discretize

prob = discretize.Problem
probu = prob.upper()
probl = prob.lower()

FILE = probl + '_main.cpp'

new = '\\n'
# The name of the IPOPT main file
f = open(FILE,'w')
```

```
f.write(‘// %s_main.cpp\n\
// Main file for use with IPOPT\n’ % (probl))

f.write(‘\
\n\
#include "IpIpoptApplication.hpp"\n\
#include "%s_nlp.hpp"\n\
\n\
// for printf\n\
#ifdef HAVE_CSTDIO\n\
# include <cstdio>\n\
#else\n\
# ifdef HAVE_STDIO_H\n\
#  include <stdio.h>\n\
# else\n\
#  error "don\’t have header file for stdio"\n\
# endif\n\
#endif\n\
\n\
using namespace Ipopt;\n\
\n\
int main(int argv, char* argc[])\n\
{\n\
\n\
  // Create a new instance of your nlp\n\
  //  (use a SmartPtr, not raw)\n\
  SmartPtr<TNLP> mynlp = new %s_NLP();\n\
\n\
  // Create a new instance of IpoptApplication\n\
  //  (use a SmartPtr, not raw)\n\
  SmartPtr<IpoptApplication> app = new IpoptApplication();\n\
\n\
  // Change some options\n\
  // Note: The following choices are only examples, they
  might not be\n\
  //       suitable for your optimization problem.\n\
  app->Options()->SetNumericValue("tol", 1e-12);\n\
  app->Options()->SetStringValue("mu_strategy", "adaptive");\n\
  app->Options()->SetStringValue("output_file", "ipopt.out");\n\
  // The following overwrites the default name(ipopt.opt) of the\n\
  // options file\n\
  app->Options()->SetStringValue("option_file_name", "%s.opt");\n\
\n\
```

```
  // Intialize the IpoptApplication and process the options\n\
  ApplicationReturnStatus status;\n\
  status = app->Initialize();\n\
  if (status != Solve_Succeeded) {\n\
    printf("%s%s *** Error during initialization!%s");\n\
    return (int) status;\n\
  }\n\
\n\
  // Ask Ipopt to solve the problem\n\
  status = app->OptimizeTNLP(mynlp);\n\
\n\
  if (status == Solve_Succeeded) {\n\
    printf("%s%s*** The problem solved!%s");\n\
  }\n\
  else {\n\
    printf("%s%s*** The problem FAILED!%s");\n\
  }\n\
\n\
  // As the SmartPtrs go out of scope, the reference count\n\
  // will be decremented and the objects will automatically\n\
  // be deleted.\n\
\n\
  return (int) status;\n\
}\n' % (probl, probu, probl,new,new,new,new,new,new,new,new,new))
f.close()
```

B.4 Makehpp.py

```
#######################################################
#
#  20 October 2009
#  Bryan A. Toth
#  University of California, San Diego
#  btoth@physics.ucsd.edu
#
#  This script writes the class header file for a C++ IPOPT
#  program defined by the vector field in the file
#  equations.txt and written by the script makecode.py.
#  The file written by this script defines a class that is
#  further described in the file written by makecode.py.
#
#  This script has been developed as part of a suite of
```

```
#  python scripts to define a dynamic parameter estimation
#  problem using the optimization software IPOPT, but is
#  generally applicable to any application needing
#  discretized derivatives of a vector field.
#
#########################################################

import discretize

prob = discretize.Problem
probu = prob.upper()
probl = prob.lower()

FILE = probl + '_nlp.hpp'

# The name of the IPOPT header file
f = open(FILE,'w')

f.write(`// %s.hpp\n\
// Header file for %s.cpp\n\
// For use with IPOPT\n' % (probl, probl))

f.write(`\n\
\n\
#ifndef __%s_NLP_HPP__\n\
#define __%s_NLP_HPP__\n\
\n\
#include "IpTNLP.hpp"\n\
#include <iostream>\n\
#include <fstream>\n\
#include <string>\n\
#include <stdlib.h>\n\
#include <cstring>\n\
\n\
using namespace std;\n\
using namespace Ipopt;\n' % (probu, probu))


f.write(`\n\n\
class %s_NLP : public TNLP\n\
{\n\
public:\n\
  /** default constructor */\n\
```

```
  %s_NLP();\n\
\n\
  /** default destructor */\n\
  virtual ~%s_NLP();\n\
\n\
  /**@name Overloaded from TNLP */\n\
  //@{\n\
  /** Method to return some info about the nlp */\n\
 virtual bool get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,\n\
               Index& nnz_h_lag, IndexStyleEnum& index_style);\n\
\n\
  /** Method to return the bounds for my problem */\n\
 virtual bool get_bounds_info(Index n, Number* x_l, Number* x_u,\n\
                  Index m, Number* g_l, Number* g_u);\n\
\n\
  /** Method to return the starting point for the algorithm */\n\
 virtual bool get_starting_point(Index n, bool init_x, Number* x,\n\
                       bool init_z, Number* z_L, Number* z_U,\n\
                       Index m, bool init_lambda,\n\
                       Number* lambda);\n\
\n\
  /** Method to return the objective value */\n\
  virtual bool eval_f(Index n, const Number* x, bool new_x,
  Number& obj_value);\n\
\n\
  /** Method to return the gradient of the objective */\n\
  virtual bool eval_grad_f(Index n, const Number* x, bool new_x,
  Number* grad_f);\n\
\n\
  /** Method to return the constraint residuals */\n\
 virtual bool eval_g(Index n, const Number* x, bool new_x, Index m,
 Number* g);\n\
\n\
  /** Method to return:\n\
   *   1) The structure of the jacobian (if "values" is NULL)\n\
   *   2) The values of the jacobian (if "values" is not NULL)\n\
   */\n\
  virtual bool eval_jac_g(Index n, const Number* x, bool new_x,\n\
             Index m, Index nele_jac, Index* iRow, Index *jCol,\n\
             Number* values);\n\
\n\
  /** Method to return:\n\
   *   1) The structure of the hessian of the lagrangian
```

```
   (if "values" is NULL)\n\
   *   2) The values of the hessian of the lagrangian
   (if "values" is not NULL)\n\
   */\n\
  virtual bool eval_h(Index n, const Number* x, bool new_x,\n\
            Number obj_factor, Index m, const Number* lambda,\n\
            bool new_lambda, Index nele_hess, Index* iRow,\n\
            Index* jCol, Number* values);\n\
\n\
  //@}\n\
\n\
  /** @name Solution Methods */\n\
  //@{\n\
  /** This method is called when the algorithm is complete so the
  TNLP can store/write the solution */\n\
  virtual void finalize_solution(SolverReturn status,\n\
   Index n, const Number* x, const Number* z_L, const Number* z_U,\n\
   Index m, const Number* g, const Number* lambda,\n\
   Number obj_value,\n\
 const IpoptData* ip_data,\n\
 IpoptCalculatedQuantities* ip_cq);\n\
  //@}\n\
\n\
private:\n\
   */\n\
  //@{\n\
  //  %s_NLP();\n' % (probu, probu, probu, probu))

for i in range(discretize.nU):
  f.write('\
  double* %s;\n\
  double* %s;\n'%(discretize.Ldata[i], discretize.Ldata[i]+'dummy'))

for i in range(discretize.nI):
  f.write('\
  double* %s;\n\
  double* %s;\n'%(discretize.Lstimuli[i], discretize.Lstimuli[i]+
  'dummy'))

#  int %s;\n' % (discretize.Ldata[i], discretize.Ldata[i]+
'dummy',discretize.Ldata[i]+'skip'))

f.write('\
```

```
\n\
  int nU;\n\
  int nP;\n\
  int nY;\n\
  int nI;\n\
  int skip;\n\
  double* K11val;\n\
  double* K11val2;\n\
  double* K11valp1;\n\
  double* dK11val;\n\
  double* dK11val2;\n\
  double* dK11valp1;\n\
  double* Xdval;\n\
  double* Xdval2;\n\
  double* Xdvalp1;\n\
  double* Xval;\n\
  double* Xval2;\n\
  double* Xvalp1;\n\
  double* Pval;\n\
  double* Ival;\n\
  double* Ival2;\n\
  double* Ivalp1;\n\
  double* Rvalue;\n\
  int Time;\n\
  double hstep;\n\
  string buffer;\n\
  string* specs;\n\
  double** bounds;\n\
  %s_NLP(const %s_NLP&);\n\
  %s_NLP& operator=(const %s_NLP&);\n\
  //@}\n\
};\n\
\n\
\n\
#endif\n' % (probu, probu, probu, probu))

f.close()
```

B.5 Makemake.py

```
import discretize

prob = discretize.Problem
probu = prob.upper()
probl = prob.lower()

nF = discretize.nF

FILE = 'Makefile'

# The name of the IPOPT main file
f = open(FILE,'w')

string = '\\'

f.write(`# Makefile for %s IPOPT problem\n\n\
##################################################################\n\
#    You can modify this example makefile to fit for your   #\n\
#     own program.  Usually, you only need to change the five #\n\
#     CHANGEME entries below. \n\
##################################################################\n\
\n\
# CHANGEME: This should be the name of your executable\n\
EXE = %s_cpp\n\
\n\
OBJS = %s_main.o %s\n' % (probl, probl, probl, string))
if nF == 0:
  f.write(`       %s_nlp.o\n' % probl)
else:
  f.write(`       %s_nlp.o %s\n' % (probl,string))
  f.write(`       myfunctions.o\n')
f.write(`\n\
# CHANGEME: Additional libraries\n\
# CHANGEME: Additional flags for compilation (e.g., include flags)\n\
ADDINCFLAGS =\n\
\n\
#SRCDIR = \n\
#VPATH = \n\
\n\
##################################################################\n\
#  Usually, you don\'t have to change anything below.  Note#\n\
#   that if you change certain compiler options, you might    #\n\
#   have to recompile Ipopt. #\n\
```

```
##############################################################\n\
\n\
# C++ Compiler command\n\
CXX = g++\n\
\n\
# C++ Compiler options\n\
CXXFLAGS = -O3 -fomit-frame-pointer -pipe -DNDEBUG -pedantic-errors
-Wimplicit -Wparentheses -Wreturn-type -Wcast-qual -Wall
-Wpointer-arith -Wwrite-strings -Wconversion -Wno-unknown-pragmas\n\
\n\
# additional C++ Compiler options for linking\n\
CXXLINKFLAGS = -Wl,--rpath -Wl,/usr/local/lib\n\
\n\
# Directory with header files\n\
IPOPTINCDIR = ${prefix}/include/coin\n\
\n\
# Directory with libipopt.a\n\
IPOPTLIBDIR = ${exec_prefix}/lib\n\
exec_prefix = ${prefix}\n\
prefix = /usr/local\n\
\n\
# Libraries necessary to link with IPOPT\n\
LIBS = -L$(IPOPTLIBDIR) -lipopt -lpthread /home/btoth/lib/
libpardiso400_GNU432_IA32.so /usr/lib/liblapack-3.so /usr/
lib/libblas-3.so -lm  -ldl  -L/usr/lib/gcc/i486-linux-gnu/4.4.3
-L/usr/lib/gcc/i486-linux-gnu/4.4.3/../../../../lib -L/lib/../lib
-L/usr/lib/../lib -L/usr/lib/gcc/i486-linux-
gnu/4.4.3/../../.. -L/usr/lib/i486-linux-gnu
-lpthread -lgfortranbegin -lgfortran -lm -lgomp -lgcc_s\n\
\n\
# Necessary Include dirs (we use the CYGPATH_W variables to allow\n\
# compilation with Windows compilers)\n\
INCL =  -I‘$(CYGPATH_W) $(IPOPTINCDIR)‘ $(ADDINCFLAGS)\n\
\n\
CYGPATH_W = echo\n\
\n\
all: $(EXE)\n\
\n\
.SUFFIXES: .cpp .c .o .obj\n\
\n\
$(EXE): $(OBJS)\n\
bla=;%s\n\
for file in $(OBJS); do bla="$$bla‘$(CYGPATH_W) $$file‘";done; %s\n\
```

```
$(CXX) $(CXXLINKFLAGS) $(CXXFLAGS) -o $@ $$bla $(ADDLIBS) $(LIBS)\n\
\n\
clean:\n\
rm -rf $(EXE) $(OBJS)\n\
\n\
.cpp.o:\n\
$(CXX) $(CXXFLAGS) $(INCL) -c -o $@ `test -f \'$<\' || echo
\'$(SRCDIR)/\'`$<\n\
\n\
\n\
.cpp.obj:\n\
$(CXX) $(CXXFLAGS) $(INCL) -c -o $@ `if test -f \'$<\';
then $(CYGPATH_W) \'$<\'; else $(CYGPATH_W)
\'$(SRCDIR)/$<\'; fi`\n' % (string, string))

f.close()
```

Bibliography

[1] H. D. I. Abarbanel. Effective actions for statistical data assimilation. *Physics Letters A*, 373:4044–4048, 2009.

[2] H. D. I Abarbanel, P. Bryant, P. E. Gill, M. Kostuk, J. Rofeh, Z. Singer, B. Toth, and E. Wong. Dynamical parameter and state estimation in neuron models. In D. Glanzman and D. Mingzhou, editors, *The Dynamic Brain: An Exploration of Neuronal Variability and Its Functional Significance*. Oxford University Press, 2011.

[3] H. D. I Abarbanel, D. Creveling, and J. Jeanne. Estimation of parameters in nonlinear systems using balanced synchronization. *Physical Review E*, 53:016208, 2008.

[4] H. D. I. Abarbanel, D. R. Creveling, R. Farsian, and M. Kostuk. Dynamical state and parameter estimation. *SIAM Journal of Applied Dynamical Systems*, 8:1341–1381, 2009.

[5] A. Arakawa. Computational design for long-term numerical integration of the equations of fluid motion: Two-dimensional incompressible flow. *J. of Comp. Physics*, 1:119–143, 1966.

[6] R. C. Aster, B. Borchers, and C. H. Thurber. *Parameter Estimation and Inverse Problems*. Elsevier Academic Press, Burlington, MA, 1987.

[7] Y. Bard. *Nonlinear parameter estimation*. Academic Press, New York, 1974.

[8] J. V. Beck and K. J. Arnold. *Parameter estimation in engineering and science*. Wiley, New York, 1977.

[9] J. M. Bower and D. Beeman, editors. *The Book of Genesis: Exploring Realistic Neural Models and the General Neural Simulation System*. Springer Verlag, New York, 1998.

[10] O. Certik. Sympy library for symbolic mathematics. Technical report, 2006.

[11] P. Courtier, J. N. Thépaut, and A. Hollingsworth. A strategy for operational implementatino of 4d-var, using an incremental approach. *Quarterly Journal of the Royal Meteorological Society*, 120:1367–1387, 1994.

[12] M. K. Cowles and B. P. Carlin. Markov chain monte carlo convergence diagnostics: A comparative review. *J. of the American Statistical Society*, 91(434):883–904, 1996.

[13] D. Creveling, P. E. Gill, and H. D. I. Abarbanel. State and parameter estimation in nonlinear systems as an optimal tracking problem. *Physics Letters A*, 372:2640–2644, 2008.

[14] P. Dayan and L. F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. MIT Press, Cambridge MA, 2001.

[15] G. Evensen. *Data assimilation: the ensemble Kalman filter*. Springer, Berlin, 2007.

[16] G. Evensen. The ensemble kalman filter for combined state and parameter estimation. *IEEE Control Systems Magazine*, 2009.

[17] M. Falcke, R. Huerta, M. I. Rabinovich, H. D. I. Abarbanel, R. C. Elson, and A. I. Selverston. Modeling observed chaotic oscillations in bursting neurons: the role of calcium dynamics and ip3. *Biol Cyber*, 82:517–527, 2000.

[18] R. M. Fano. *Transmission of Information: A Statistical Theory of Communication*. Wiley, New York, 1961.

[19] R FitzHugh. Impulses and physiological states in theoretical models of nerve membrane. *Biophysics J.*, 1:445–466, 1961.

[20] R FitzHugh. A kinetic model of the conductance changes in nerve membrane. *J. Cell. Comp. Physiol.*, 66:111–118, 1965.

[21] R FitzHugh. Mathematical models for excitation and propagation in nerve. In H. P. Schwan, editor, *Biological Engineering*. McGraw Hill, New York, 1969.

[22] P.E. Gill, W. Murray, and M.A. Saunders. Snopt: An sqp algorithm for large-scale constrained optimization. *SIAM Review*, 47(1):99–131, 2005.

[23] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol.*, 10:500–544, 1952.

[24] D. Johnston and S. M. Wu. *Foundations of Cellular Neurophysiology*. MIT Press, Cambridge MA, 1995.

[25] N. G. Van Kampen. *Stochastic Processes in Physics and Chemistry, Third Edition*. North Holland, 2007.

[26] C. Koch. *Biophysics of Computation: Information Processing in Single Neurons*. Oxford University Press, New York, 1999.

[27] C. Koch and I. Segev, editors. *Methods in Neuronal Modeling*. MIT Press, Cambridge MA, 1998.

[28] F. X. Le Dimet and O. Talagrand. Variational algorithm for analysis and assimilation of meteorological observations: Theoretical aspects. *Tellus*, 38A:97–110, 1986.

[29] B. C. Levy. *Principles of signal detection and parameter estimation*. Springer, New York, 2008.

[30] E. N. Lorenz. Deterministic nonperiodic flow. *J. Atmos. Sci.*, 20:130–141, 1963.

[31] E. N. Lorenz. Predictability–a problem partly solved. In *Proceedings of the seminar on predictability, European center for medium range weather forecasting*, volume 1, pages 1–18. Reading, Berkshire, UK, 1996.

[32] E. N. Lorenz and K. A. Emanuel. Optimal sites for supplementary weather observations: Simulation with a small model. *J. Atmos. Sci.*, 55:399–414, 1998.

[33] A. Maybhate and R. E. Amriktar. Use of synchronization and adaptive control in parameter estimation from a time series. *Physical Review E*, 59:284, 1999.

[34] T. McKenna, J. Davis, and S. F. Zornetzer, editors. *Single Neuron Computation*. Academic Press, Boston, 1992.

[35] C. Morris and M. Lecar. Voltage oscillations in the barnacle giant muscle'. *Biophys. J.*, 71:3030–3045, 1981.

[36] J. S. Nagumo, S. Arimoto, and S. Yoshizawa. An active pulse transmission line simulating nerve axon. *Proc. IRE*, 50:2061–2070, 1962.

[37] H. Nijmeijer. A dynamical control view on synchronization. *Physica D*, 154:219–228, 2001.

[38] T. Nowotny, R. Levi, and A. I. Selverston. Probing the dynamics of identified neurons with a data-driven modeling approach. *PLoS ONE*, 3(e2627), 2008.

[39] U. Parlitz, L/ Junge, W. Lauterborn, and L. Kocarev. Experimental observation of phase synchronization. *Physical Review E*, 43:2115, 1996.

[40] L. M. Pecora and T. L. Carroll. Synchronization in chaotic systems. *Physical Review Letters*, 64:821–824, 1990.

[41] A. Pikovsky, M. Rosenblum, and J. Kurths. *Synchronization. A universal concept in nonlinear sciences*. Cambridge: Cambridge University Press, 2001.

[42] J. C. Quinn and H. D. I. Abarbanel. State and parameter estimation using monte carlo evaluation of path integrals. *Quarterly Journal of the Royal Meteorological Society*, 136:1855–1867, 2010.

[43] J. C. Quinn, P. H. Bryant, D. R. Creveling, S. R. Klein, and H. D. I. Abarbanel. Parameter and state estimation of experimental systems using synchronization. *Phys. Rev. E*, 80:016201, 2009.

[44] O. Schenk, M. Bollhoefer, and R. Roemer. On large-scale diagonalization techniques for the anderson model of localization. *SIAM Review*, 50:91–112, 2008.

[45] O. Schenk, A. Waechter, and M. Hagemann. Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization. *J. of Comp. Optimization and Applications*, 36(2–3):321–341, 2007.

[46] A. Selverston, D. F. Russell, J. P. Miller, and D. G. King. The stomatogastric nervous system: Structure and function of a small neural network. *Prog Neurobiol*, 7:215–290, 1976.

[47] H. W. Sorenson. *Parameter estimation: principles and problems*. M. Dekker, New York, 1980.

[48] W. J. H. Stortelder. *Parameter estimation in nonlinear dynamic systems*. Centrum voor Wiskunde en Informatica, Amsterdam, 1998.

[49] S. H. Strogatz. *Nonlinear Dynamics and Chaos*. Westview Press, Cambridge MA, 1994.

[50] A. Tarantola. *Inverse problem theory: Methods for data fitting and model parameter estimation*. Elsevier Science Pub. Co. Inc., New York, 1987.

[51] A. Tarantola. *Inverse problem theory and methods for model parameter estimation*. SIAM, Philadelphia, 2005.

[52] B. A. Toth, M. Kostuk, C. D. Meliza, D. Margoliash, and H. D. I. Abarbanel. Dynamical estimation of neuron and network properties 1: Variational methods. *Biocybernetics and Biomedical Engineering*, 2011.

[53] H. U. Voss, J. Timmer, and J. Kurths. Nonlinear system identification from uncertain and indirect measurements. *International Journal of Bifurcation and Chaos*, 14:1905–1933, 2004.

[54] A. *Wächter* and L. T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–27, 2006.

[55] Jean Zinn-Justin. *Quantum Field Theory and Critical Phenomena*. Clarendon Press, Oxford, 2002.