Spring 2007

Code-level modeling of the Hodgkin -Huxley neuron model using an open source version of SPICE

Anthony Stuart Carver
Louisiana Tech University

Code-Level Modeling of the Hodgkin-Huxley Neuron Model Using an Open Source Version of SPICE

By

Anthony Stuart Carver, BS, MS

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

COLLEGE OF ENGINEERING AND SCIENCE
LOUISIANA TECH UNIVERSITY

May 2007

LOUISIANA TECH UNIVERSITY

THE GRADUATE SCHOOL

April 18, 2007
Date

We hereby recommend that the dissertation prepared under our supervision by Anthony S. Carver entitled Code-Level Modeling of the Hodgkin-Huxley Neuron Model Using an Open Source Version of SPICE be accepted in partial fulfillment of the requirements for the Degree of PhD - Engineering

Robert Sylvester JPP
Supervisor of Dissertation Research

Head of Department
Engineering and Science
Department

Recommendation concurred in:

Advisory Committee

Approved:
Director of Graduate Studies

Approved:
Dean of the Graduate School

Dean of the College

GS Form 13
(5/03)

ABSTRACT

There have been numerous studies presented in the literature demonstrating proof of principle neural-electronic circuitry. Some of these studies involve simulations of neural detection using synthetic electronic circuitry, while others involve simulations of neural excitation using external electronics. A common feature of these studies is the simplicity of the overall circuit topology. Some of these studies implement the circuit equations in conventional numerical ordinary differential equation solvers. This process involves the algebraic manipulation of the circuit equations which is a tedious process for all but the simplest circuit topologies. As the overall complexity of the network topology increases, the numerical solver approach quickly becomes intractable necessitating an alternate implementation strategy. SPICE implementations of the Hodgkin-Huxley neuron model have sought to remedy this problem. There have been multiple studies associated with implementing the Hodgkin-Huxley model in the open source circuit simulator, SPICE. In this dissertation, a novel implementation of a portable SPICE device model developed using the Hodgkin-Huxley active membrane model is implemented using the code-level modeling functionality of an open source version of SPICE. The model is validated by comparison with standard Hodgkin-Huxley model simulations including gating variable dynamics simulations, accommodation, anode-break excitation, and others. A further validation study is carried out demonstrating two blocking phenomenon described in the literature. The device model fully parameterizes

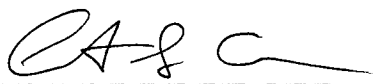
the Hodgkin-Huxley membrane model to include temperature, internal and external concentrations used in the Nernst equations, and other user specified parameter values. This parameterization allows for making changes to the underlying neuron model rapidly and with minimal implementation complexity.

The novelty and robustness of the modeling approach described herein is based on the ease of implementation. A wide variety of active membranes can be simulated using this code model approach. These biologically realistic components can be integrated with artificial electronic components allowing for the simulation of hybrid neural-electronic circuitry under the SPICE simulation platform. These types of hybrid circuit simulations are not currently achievable using other neural simulators such as NEURON or GENESIS. While this implementation uses the Hodgkin-Huxley neuron model with its known limitations, the process of developing the device model can be used to implement any neuron model which can be described mathematically.

APPROVAL FOR SCHOLARLY DISSEMINATION

The author grants to the Prescott Memorial Library of Louisiana Tech University the right to reproduce, by appropriate methods, upon request, any or all portions of this Dissertation. It is understood that "proper request" consists of the agreement, on the part of the requesting party, that said reproduction is for his personal use and that subsequent reproduction will not occur without written approval of the author of this Dissertation. Further, any portions of the Dissertation used in books, papers, and other works must be appropriately referenced to this Dissertation.

Finally, the author of this Dissertation reserves the right to publish freely, in the literature, at any time, any or all portions of this Dissertation.

Author 

Date 04/20/2007

GS Form 14
(5/03)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to acknowledge the people who have provided me with support, encouragement, and the opportunity to complete this work. First, I would like to say a HUGE thank you to my wife, Teri, for her continued support throughout the years not only in this endeavor, but life in general. Her unconditional love and support is the greatest gift I could ever receive. Without you walking beside me daily, this accomplishment, and life as a whole, is far less meaningful. For my kids, Ashlee and Logan, I hope I didn't take too much time away from you. I can not thank you guys enough for your support and understanding. You are my motivation to succeed! To my Mom and Dad, Faye and Noble Carver, I could not ask for better parents. Your support and encouragement have helped me stay focused and motivated. To all my extended family that has continuously supported me, thank you. To my advisor, Dr. Robert Szlavik, I am so thankful for our chance meeting in the electronic circuits course. It is through your guidance and mentoring that this work was possible. To my research partner, Frank Jenkins, it has been an honor working with you in the Lab. You have been a tremendous help, and I doubt I will ever find someone to rival your ability to put away the Chinese food! Finally, to Air Force Major Don Copsey, you are the best "used car salesman" I have ever known! If not for you, this opportunity would never have happened. This is all, your fault! ☺

CHAPTER 1

INTRODUCTION

There have been several studies presented in the literature that demonstrate proof of principle neural-electronic circuitry [1-5]. Some of these studies include simulations of neural detection using synthetic electronic circuitry [2;5]. Additional studies include simulations of neural excitation using external electronics [1;3]. The simplicity of the overall circuit topology is a common feature of the simulations presented in the above literature. The approach adopted, in at least some of these studies, involves implementation of the circuit equations in conventional numerical ordinary differential equation solvers [2;3;6]. Algebraic manipulation of the circuit equations involves rewriting these equations in a form whereby the first derivatives of each of the differential equations is isolated on one side of the equation. This manipulation of the circuit equations can be a tedious process for all but the simplest circuit topologies. As the overall complexity of the network topology becomes more involved and as the number of nodes in the system increases, the conventional numerical solver approach rapidly becomes intractable necessitating an alternate implementation strategy. It is quite likely that the difficulty associated with implementation of more involved network simulations has been a principal limiting factor in advancements in the state of the art of neural-electronic circuit integration.

There have been various SPICE neuron models presented in the literature [7;8]. These models are based on the Hodgkin-Huxley active membrane model [9-13]. Models developed in SPICE basically fall into one of two categories. The first category involves modifying the source of the actual SPICE code. One such SPICE model implemented using Hodgkin-Huxley dynamics was documented in the literature by Bove et al [7]. The model was implemented by altering the source code of the actual SPICE 2G software. This approach was termed a "built in" model. The built in process required the addition of five new subroutines and the modification of four existing SPICE2G subroutines. Unlike previous work, this approach directly implemented the Hodgkin-Huxley equations. However, the dynamics of the Hodgkin-Huxley model were modified ad hoc into the SPICE source code. This process did not allow for updates as new versions of SPICE were released. Also, no parameterization of any Hodgkin-Huxley variables was documented in the literature. This fact made the software less user friendly requiring changes to the source code as well as recompilation in order to make changes to the simulation values.

There was a second issue with the "built in" model. It exhibited a resting membrane potential inconsistent with physiological observations. The model used a biologically unrealistic value for the resting membrane potential of 0 mV, whereas realistic resting membrane potentials exist in the -60 to -70 mV range.

The second category of SPICE models involves using passive devices within SPICE (resistors, capacitors, etc) as well as active, nonlinear polynomial sources to develop the equivalent circuit model for the Hodgkin-Huxley Model. The earlier models had two issues concerning their utility. The earlier models exhibited the resting

membrane potential inconsistent with physiological observations [8;14]. In the latest study of this type of model, a modified version of a previous SPICE based neuron model was detailed which incorporates the non-linear exponential functions that describe the gating variable rate constants' dependence on the transmembrane potential [8;15]. This model also demonstrated more physiologically relevant electrical behavior of the simulated neuron by demonstrating membrane potential variations and levels that are consistent with the expected physiological behavior of electrically active cell membranes. While this study improved on the previous models, the process was complicated due to the use of a SPICE subcircuit to implement the m, h, and n gating variables (gating variables are discussed in chapter 2). This subcircuit added approximately 40 lines of SPICE code in the netlist file. This model also had no parameterization associated with the temperature or other Hodgkin-Huxley variables.

The objective of this study was to create a platform portable, fully parameterized device model based on the full Hodgkin-Huxley equations. This parameterization not only includes the actual Hodgkin-Huxley model parameters but the variables of the Nernst equations as well as temperature and cell dimensions for the cylindrical nerve cell modeled. The final objective of the study was to implement the model in a way which allowed for compartmental axon simulations by tying multiple neurons in sequence to build more complicated axonal structures.

In this study, the Hodgkin-Huxley model was specifically chosen with an understanding of its shortfalls. The Hodgkin-Huxley model is implemented using the code model approach introduced by Cox and colleagues [16]. While the Hodgkin-Huxley model was the model-of-choice for this implementation, it is important to note

any mathematical neuron model could be chosen and implemented using the code-model approach in this dissertation. This approach allows for model variables to be programmed into the user created device model, and allows for simulations using different parameters over multiple simulation runs without having to recompile the source code. Temperature dependence is tied to the SPICE .option variable "temp" which is set within the SPICE circuit file. This flexible approach is demonstrated in this study in the context of two groups of validation simulations. The first demonstration involves simulating multiple phenomenon associated with the Hodgkin-Huxley model. These simulations are then compared to documented values found in the literature to validate the models ability to reproduce known data. The second demonstration involves simulating two types of thermal blocking of action potentials described in the literature. The first blocking simulation shows the basic thermal block as first defined experimentally by Hodgkin and Katz and described for the Hodgkin-Huxley equations in Weiss [17;18]. This simulation shows the thermal blocking phenomenon that exists between 22-23°C. Second, we demonstrate our device models ability to simulate the direct current axonal conduction block reported by Bhadra [19].

This dissertation is broken up into eight chapters including the introduction. Chapter two discusses the cellular specifics necessary to understand the action potential and the Hodgkin-Huxley model. Chapter three describes the action potential itself, and introduces key characteristics of the action potential. Once the necessary background is reviewed, the Hodgkin-Huxley model is introduced. First, a brief history of the Noble Prize winning papers written by Hodgkin and Huxley is given followed by the mathematical description of the actual model. Chapter four also defines key aspects of

the model used for validation later on. Chapter five gives a history of the SPICE circuit simulator and the different versions used in this dissertation. Chapter six provides an in-depth introduction to the XSPICE Code Model Toolkit which provides the functionality to build the new device model. Chapter seven documents the files used to create the actual device model. Chapter seven also lays out the parameterization of the Hodgkin-Huxley model and the associate Nernst equations. Finally, Chapter eight provides a thorough validation of the device model by comparing the graphical results of device model simulations to graphical values found in the literature. Actual graphs from Weiss are provided in appendix C for comparison by the reader [18].

CHAPTER 2

CELLULAR SPECIFICS

Before any discussion of the action potential or the Hodgkin-Huxley model can begin, the environment in and around the cell, as well as the makeup of the cellular membrane must be explained. How the cell maintains its osmotic and ionic balance are important details in describing the potentials involved in the resting and active states. The Nernst potential describes the potential associated with a single ion. The Donnan equilibrium describes the concentration conditions for equilibrium of two ions when two ions are at equilibrium across a cell membrane, while the Goldman equation describes the membrane potential with regards to multiple ions and their relative permeabilities.

2.1 Intracellular and Extracellular Makeup

Any study of how nerve cells produce action potentials requires a discussion concerning the makeup of both extracellular and intracellular fluids around and in those cells outside and inside of the cell. The human body's weight is made up of approximately 75% water. This water is distributed at roughly 55% inside cells and 45% outside cells [20]. The water outside cells constitutes the extracellular fluid or ECF; while, the water inside cells constitutes the intracellular fluid or ICF. Different organic

and inorganic materials are found in both the ECF and ICF in varying concentrations. While other materials are present in both the ECF and ICF, the main materials of importance for this discussion are listed in table 2.1 [20]. A group of anions made up of proteins, amino acids and inorganic ions are listed together as X^- in order to show equilibrium between the ECF and ICF.

Table 2.1. Ionic concentrations for ICF and ECF [20].

	ICF Concentration (mM)	ECF Concentration (mM)
Na^+	12	120
K^+	125	5
Cl^-	5	125
X^-	108	0

The ECF has a high concentration of positive sodium ions as well as negative chloride ions. There is also a small concentration of positive potassium ions which will play a role in action potential generation discussed later. The ICF has the opposite makeup of the ECF. In the ICF, high concentrations of positive potassium ions are present, but the concentrations of positive sodium and negative chloride ions are relatively low. These concentrations are fundamental to the action potential generation and the waveform created during the firing of nerve cells.

2.2 Cellular Membrane

Understanding the cellular membrane and how it functions is imperative to understanding the action potential and how it is generated. A cell's membrane is composed of to two layers of lipids and collectively is referred to as a lipid bilayer. These lipid molecules are phospholipids which have a polar hydrophilic end and a nonpolar hydrophobic end. These molecules form a bilayer in which the hydrophilic ends are oriented outward and the hydrophobic ends are oriented towards each other as shown in Figure 2.1.

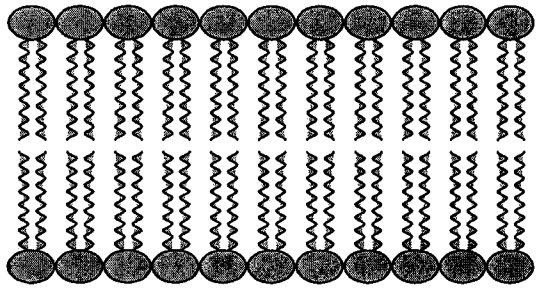


Figure 2.1. An example of a lipid bilayer cell membrane. The hydrophobic ends are oriented toward each other leaving the hydrophilic ends pointing outward and inward forming a two molecule thick cell membrane.

The lipid bilayer is not the only component of the cellular membrane. Dispersed among the bilayer structure are several functional molecules that lend into the functionality of the cellular membrane which are fundamental in the generation of the action potential. One such type of molecule are the channels made of proteins running through the entire membrane that form aqueous pores through the membrane [20]. These voltage gated channels act as passage ways for ions and other molecules. The channels can be general in nature allowing multiple ions and molecules through, or they can be very selective allowing only specific ionic species to pass through. The state of the

channel is determined by whether the channel is "open" or "closed". These channels have gating particles associated with binding sites which determine the state of the channel. In the case of binding site channels, the presence of a neurotransmitter molecule, such as acetylcholine, open the channel when present at the binding site [20].

For the Hodgkin-Huxley model, there are two channels of interest shown in figure 2.2. The sodium channel is a double gated channel consisting of the fast acting "m" gate and the slow acting "h" gate, while the potassium channel has a single slow acting "n" gate. These gates open and close based on the dynamics of the environment during action potential cycle. Specifics on how this affects action potential generation will be covered in Chapter 3.

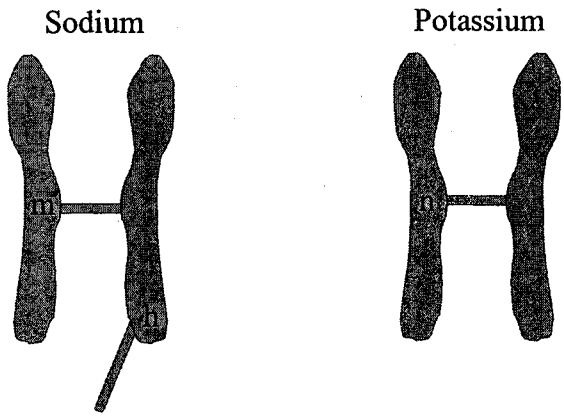


Figure 2.2. Sodium (m/h) and potassium (n) gated ion channels.

2.3 Cellular Functionality

The action potential in animal nerve cells is a complicated phenomenon requiring several conceptual pieces to explain the cellular environment such as osmotic balance;

multiple permeable ions of different size; active transport of certain ions; and the balance maintained between concentration and electrical gradients. The following sections discuss the specifics which allow neuronal cells to produce an action potential.

2.3.1 Osmotic Balance

The cellular membrane is a key component in maintaining osmotic balance between the ICF and ECF. To understand this idea, definitions for osmolarity and osmosis are needed. Osmolarity describes the amount of dissolved substances within a solution. For example, 1 mole of a substance dissolved into a liter of solution would result in an osmolarity of 1 osmolar (1 Osm). Osmosis can be defined as the movement of water down its concentration gradient [20]. Osmotic balance occurs when the concentration of solute on one side of a permeable divider are equal to the concentrations on the other side of the divided container. An example of osmotic balance could include the same container from above with a container separated by a permeable divider being filled with solutions of different osmolarity of the same dissolved particle. The side with a higher osmolarity will contain a larger amount of dissolved particles, and because of this, it will contain a smaller amount of water. The water from the lower osmolar solution will also diffuse down its concentration gradient into the compartment with the higher osmolar solution. This process will continue until both sides of the container contain solutions of equal osmolarity.

2.3.2 Diffusion Potentials

The above discussion on osmotic balance is not the only factor determining where cell equilibrium will be reached. Diffusion potentials also are important in the cell

reaching its equilibrium state. In section 2.3, the particles dissolved in the solution were non-charged particles. In real cells, the particles dissolved in solution, mainly sodium, chloride, and potassium are charged. These particles, also called ions, are either positively charged ions called cations, or negatively charged ions called anions [20]. A diffusion potential is present when two or more ions are moving down their concentration gradient as discussed in the last paragraph. The diffusion potential will be created due to the different sizes of the ions in question. A good example of a situation which results in a diffusion potential is given in Matthews [20]. It involves two containers separated by a membrane permeable to both sodium (Na^+) and chloride (Cl^-). One side of the container is filled with an aqueous 10% NaCl solution, while the other side is filled with a 50% NaCl aqueous solution. Na^+ is a larger ion which moves down the concentration gradient slower than the smaller Cl^-. Because Cl^- moves faster down the concentration gradient, the concentration of Cl^- on the less concentrated side will increase faster than the concentration of Na^+ creating a negative charge in that compartment. As the less concentrated compartment becomes more negative, the negative charge begins to repel the negative Cl^- ions slowing its diffusion. The negative charge also begins to speed the larger Na^+ ion. The diffusion potential will increase to the point where the slowing of the Cl^- ion counter acts the smaller size of the Cl^- ion, and the diffusion rates of both Na^+ and Cl^- become the same. This diffusion potential at equilibrium for a particular ion is known as the ion's equilibrium or Nernst potential.

2.3.3 Equilibrium Potential – Nernst Equation

In section 2.3.2, both ions are able to diffuse across the barrier dividing the two compartments. With this situation, equilibrium is eventually reached where the

concentrations of each ion are equal in each compartment and there is no potential between the compartments. If, however, the membrane is made permeable to only one of the ions, equilibrium can only be reached when the repulsive force of the electrical potential in one compartment due to the ionic movement negates the diffusion between the compartments. The value of this electrical potential across the barrier can be calculated from the Nernst equation.

$$V_{Ion} = \left(\frac{RT}{F}\right)\log\left(\frac{C^{O}_{Ion}}{C^{i}_{Ion}}\right) \qquad 2.1$$

V_{Ion} is the Nernst potential for the specific ion; R is the molar gas constant; T is absolute temperature and F is Faraday's constant. The variables C^{O}_{Ion} and C^{i}_{Ion} are the extracellular and intracellular ionic concentrations respectively [18]. It is important to note that the Nernst equation only applies to one ion at a time, and the ion must be a permeable ion [20].

2.3.4 Space Charge Neutrality

The variables C^{O}_{Ion} and C^{i}_{Ion} in the Nernst equation are the initial concentrations of the ion in question which are assumed to be static. The initial concentrations are used based on the idea of space charge neutrality. Also known as the principle of electrical neutrality, space charge neutrality refers to the approximation that under biological conditions, the concentration of cations and anions within a compartment must be equal [20]. This approximation allows for the fact that while the different ions move between compartments to create the diffusion potentials and ultimately the settling at the Nernst equilibrium potential, the number of ions required to affect the transmembrane potentials found in biology is miniscule compared to the total amount of ions in the intracellular and

extracellular fluids. Therefore, there is no affect on the concentration gradients of ions associated with the diffusion potentials.

2.3.5 Donnan Equilibrium

While the Nernst equation gives the equilibrium potential across a permeable barrier for a specific ion, the Donnan equilibrium gives the conditions necessary for equilibrium in a system where two ions are permeable [20]. The Donnan equilibrium arises when two compartments separated by a permeable membrane contains two ions capable of moving across the membrane. If equilibrium is to be reached, the electrical potential and concentration gradients between the two compartments must balance. This implies that the Nernst potentials for both ions would be equal. Setting the Nernst equations for each ion equal gives

$$V_{Ion1} = \left(\frac{RT}{F}\right)\log\left(\frac{C^{O}_{Ion1}}{C^{i}_{Ion1}}\right) = V_{Ion2} = \left(\frac{RT}{F}\right)\log\left(\frac{C^{O}_{Ion2}}{C^{i}_{Ion2}}\right).$$

Simplifying the above equation and moving the negative valence minus sign inside the parentheses of the log term yeilds

$$\left(\frac{C^{O}_{Ion1}}{C^{i}_{Ion1}}\right) = \left(\frac{C^{i}_{Ion2}}{C^{O}_{Ion2}}\right)$$

or,

$$\left[C^{O}_{Ion1}\right]\left[C^{O}_{Ion2}\right] = \left[C^{i}_{Ion1}\right]\left[C^{i}_{Ion2}\right].$$

This relationship states that in order for equilibrium to be reached between two ions separated by a permeable divider, the product of the two ions concentration on one side must equal the concentration of the two ions on the other side of the divider [20].

2.3.6 Goldman Equation

The Nernst equation defines the equilibrium potential for a single ion. The Donnan equilibrium equation considers two ions and what must occur for the two ions to reach equilibrium between two compartments separated by a permeable membrane. The Goldman Equation,

$$E_m = \frac{RT}{F} \ln\left(\frac{p_K\left[K^+\right]_O + p_{Na}\left[Na^+\right]_O + p_{Cl}\left[Cl^-\right]_i}{p_K\left[K^+\right]_i + p_{Na}\left[Na^+\right]_i + p_{Cl}\left[Cl^-\right]_O} \right),$$

describes the situation where multiple ions with different Nernst equilibrium potentials, ionic concentrations, and ionic permeabilities are present [20]. A simplified version of the Goldman equation is

$$E_m = 58mV \log\left(\frac{\left[K^+\right]_O + b\left[Na^+\right]_O}{\left[K^+\right]_i + b\left[Na^+\right]_i} \right).$$

This version evaluates RT/F at room temperature, converts from ln to log, and expresses the results in millivolts. As an approximation, the Chloride term can be dropped. This approximation is possible because of chloride's insignificant contribution to the resting potential of a cell, which is true for most nerve cells. The equation also expresses the relative permeability of sodium in relation to the permeability of potassium [20]. Using the common value for b of 0.02, which is a 50 times greater permeability of potassium compared to sodium, it can be seen why the resting potential of a typical cell (-72 mV) is closer to the Nernst potential of potassium (-80 mV) than the Nernst potential of sodium (+58 mV) [20].

CHAPTER 3

THE ACTION POTENTIAL

The membrane potential of a cell is defined by the voltage difference across the cell membrane [20]. Electrically excitable cells are those cells which produce a change of membrane potential when a current of sufficient magnitude is passed through its cellular membrane. Once the current of sufficient magnitude is passed through the membrane, an electrical chain reaction occurs producing what is known as an action potential [18]. The cellular resting potential is defined by a dynamic equilibrium between the intracellular and extracellular environment of the cell where the net currents flowing across the cell membrane is equal to zero [21]. The value of the resting membrane potential is negative due to the convention that the outside of the cell is taken as the reference point and the intracellular potential is negative compared to the outside of the cell. Common resting potentials include values ranging from -30mV to -90mV [21]. There are six basic characteristics of the action potential that lead to the different cellular and ionic changes throughout the life of a single action potential cycle. The action potential is divided into four basic parts each having substantially different segments. These segments are the resting phase, depolarization phase, repolarization phase, and the undershoot phase. Another characteristic that the action potential exhibits involves the refractory period at the end of the action potential. During this time interval, the nerve cell is incapable of

firing another action potential. Finally, it is the propagation of the action potential down multiple nerve cells grouped together in nerve trunks that allows efferent and afferent signals to propagate to and from the central nervous system.

3.1 Properties of the Action Potential

Specific properties of the action potential lead to its ability to produce the membrane potential necessary to carry signals throughout the neuronal structure. These properties can be broken down into six specific action potential characteristics [20]. These characteristics are:

1. **Action potentials begin with a depolarization of the membrane potential.** A depolarization event happens when the inside of the nerve cell becomes less negative than the outside. This event normally happens due to external stimuli (i.e. stretching of a muscle or the activation of a neighboring nerve cell).

2. **The threshold level for the nerve cell must be reached before the action potential will fire.** There is a certain level of depolarization required to elicit an action potential. Anything less than this threshold will not cause an action potential to develop. For a typical neuron with a resting potential around -70 mV, a 10-20 mV depolarization is required to reach the threshold and start the action potential process.

3. **Action potentials are all-or-nothing.** Once a neuron reaches the threshold required to start the action potential process, it will run to completion.

4. **Action potentials exhibit "perfect reproductiveness".** The action potential, once started, propagates throughout the neuronal chain it is attached to without

losing "signal quality". That is, the signals amplitude remains the same for the duration of propagation that was initiated by the initial firing.

5. **The membrane potential reverses sign at the peak of the action potential.** The action potential overshoots zero becoming positive at its peak. After this, the inside of the cell repolarizes towards the normal negative resting potential. This repolarization causes an undershoot of the resting potential as the ionic concentrations move back to the resting state.

6. **Neuronal cells cannot fire again until a set time called the refractory period has elapsed.** This refractory period ensures that once an action potential has traversed a section of neurons, no signal can traverse that section in the reverse direction for a period of time. However, if a stimulus is initiated in the middle of an electrically long cell, an action potential will be generated in both directions away from the initiation site.

Most neuronal cell refractory periods lasts approximately 1 msec which limits the number of action potentials to around 1000 per second. These six properties lead to the description of the action potential broken down into four distinct periods of action potential development. These phases are the resting, depolarization, repolarization, and refractory phases.

3.2 Resting Phase

The first phase in the action potential process is the resting phase. This phase is the steady state for the nerve cell. Under normal conditions, with no outside stimulus

directed at the cell, the resting potential for a cell is defined somewhere between the Nernst potential for sodium (+58 mV) and the Nernst potential for potassium (-80 mV), which for a typical cell is around -72 mV. Potassium's increased permeability compared to sodium drives the resting potential closer to its value verses the value for sodium. During the resting phase, the fast acting "m" gate is closed; the slow acting "h" gate is open, and the slow acting "n" gate is closed. This phase is illustrated in Figure 3.1.

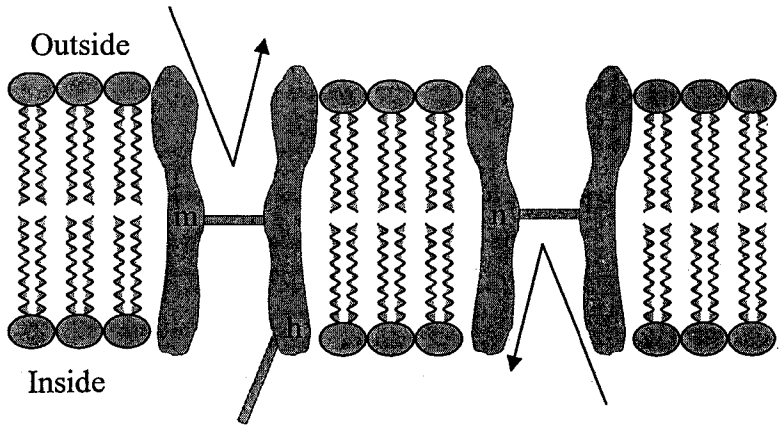


Figure 3.1. Illustration representing the resting phase before action potential initiation.

3.3 Depolarizing Phase

It is during the depolarizing phase that the properties discussed in 3.1 begin to manifest themselves. The first property, "cell depolarization", initiates the depolarization phase. In order for an action potential to start, a net depolarization in side the cell must occur. Under resting conditions, the "m" gate in the sodium channel and the "n" gate in the potassium channel are closed, maintaining the resting potential. The threshold property is necessary for the action potential to start. This level of depolarization starts a

chain of events which leads to the "all or nothing" nature of the action potential discussed in property number three. It is the depolarization of the cell membrane which opens the fast "m" sodium gate, allowing sodium ions into the cell. This process continues to depolarize the cell further, leading to the peak of the action potential; however, at the same time the "m" gate opens, the slow acting "h" gate begins to close, and the slower acting potassium "n" gate begins to open [20]. Figure 3.2 illustrates the state of the sodium and potassium gates during the depolarization phase.

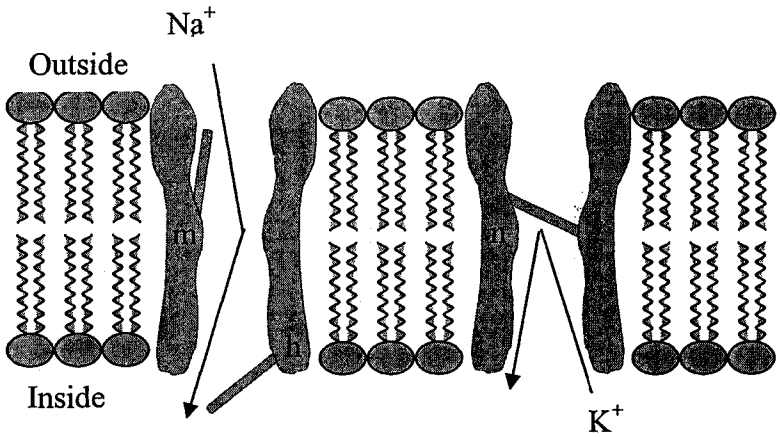


Figure 3.2. Illustration representing the depolarization phase during action potential generation.

3.4 Repolarization Phase

The transition between the depolarization and repolarization phase is associated with the fifth property of an action potential. Around 1-2 milliseconds after the "m" gates open, the "h" gates finally close stopping the influx of sodium ions into the cell. Around this time, the potassium "n" gate opens allowing for the efflux of potassium ions which begins to repolarize the nerve cell [20]. The "n" gate will remain open until the

membrane potential returns to the resting potential. Figure 3.3 illustrates the state of the sodium and potassium gates during the repolarization phase.

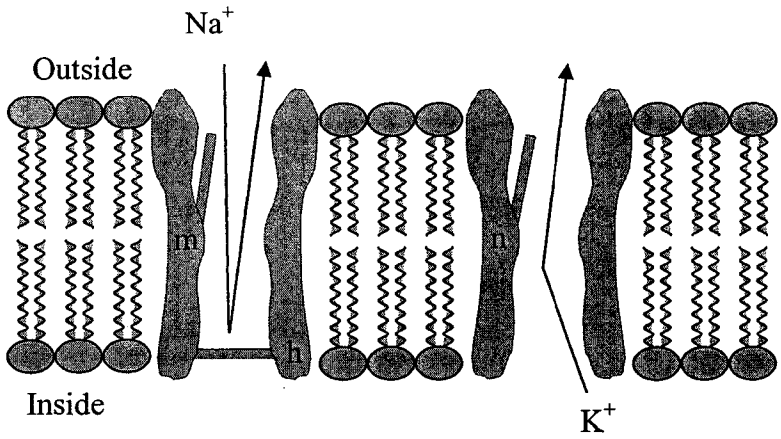


Figure 3.3. Illustration representing the repolarization phase during action potential generation.

3.5 Undershoot Phase

At the end of the repolarization phase, the sodium "h" gates have closed allowing the sodium level in the cell to normalize back to its resting value. However, the potassium "n" gates are still open allowing for potassium ions to leave the cell. These events hyperpolarize the inside of the cell making it more negative than its resting potential. This phase is know as the undershoot phase. Figure 3.4 illustrates the state of the sodium and potassium gates during the undershoot phase.

K+

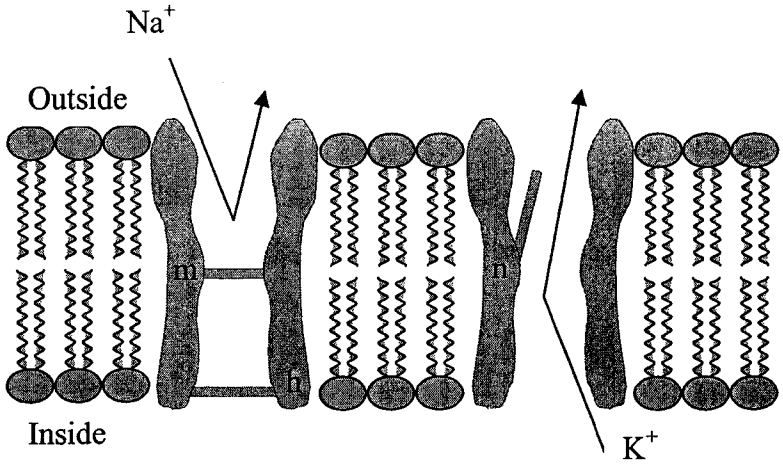


Figure 3.4. Illustration representing the undershoot phase during action potential generation.

3.6 Refractory Period

The sixth property of action potentials is associated with the cells refractory period. Once the slow acting "h" sodium gates close, a subsequent action potential in the same cell is prohibited due to the inability of sodium ions to enter the cell and begin the depolarization phase again. The time it takes the sodium "h" gates to reopen and therefore allow a new action potential to begin is known as the refractory period after the current action potential has ended [21].

3.7 Propagation

In order for communication between the brain and other parts of the body, the nerve fibers of the afferent and efferent pathways must be able to pass action potentials along long distances. The action potential in a nerve fiber can be described as a self

propagating wave [20]. Once the 10-20 mV depolarization of the initial nerve cell in a nerve fiber occurs, the depolarization of that cell affects the surrounding cells in the nerve fiber. The threshold required for the subsequent nerve cells within a nerve fiber to begin an action potential is reached due to the previous cells depolarization allowing the action potential to propagate the length of the nerve fiber. Action potentials only propagate in one direction from one nerve cell to another through a nerve fiber due to the refractory period of a cell after it has fired and began the depolarization of the next nerve cell in the chain. This propagation from nerve cell to nerve cell down a nerve fiber describes the fourth property of action potentials, "perfect reproductiveness".

CHAPTER 4

HODGKIN-HUXLEY MODEL

Hodgkin and Huxley's measurement and subsequent model formulation for the action potential in the giant axon of the Loligo squid was ground breaking work. They received the Nobel Prize in 1963 for their work in both systematically measuring the action potential characteristics in the squid as well as developing the mathematical theory for their model [9-13]. Hodgkin-Huxley theory explains the relationship, within a patch of axonal membrane, between ionic currents (sodium, potassium and leakage), capacitive currents, and the membrane current density. The model defines the relationship between the Nernst potential for the sodium and potassium ions and the ionic conductances, with respect to time and membrane potential scaled to the particular gating values for the individual ionic conductances. These gating values are determined using the rate constant equations fit by Hodgkin and Huxley to analytical expressions that are dependent on the membrane potential. After the theory was developed, it was confirmed that certain known aspects of the action potential are consistent with the theoretical predictions from the model. Repetitive activity, accommodation, anode-break excitations, subthreshold oscillations, and the effect of temperature on the action potential are described theoretically by the Hodgkin-Huxley model.

4.1 History

The Hodgkin-Huxley model is the culmination of the work of A.L Hodgkin and A.F. Huxley in the early 1950s. Hodgkin and Huxley wrote five papers documenting the experiments used to discern the characteristics and mechanisms of ion movement in nerve cells during action potential generation. All five papers were published in the Journal of Physiology. Hodgkin and Huxley were awarded the Nobel Prize in Physiology or Medicine 1963, "for their discoveries concerning the ionic mechanisms involved in excitation and inhibition in the peripheral and central portions of the nerve cell membrane". The first paper, "Measurement of Current-Voltage Relations in the Membrane of the Giant Axon of Loligo", describes the steady state conditions for the cellular membrane and the techniques to be used in the subsequent papers for their analysis [13]. The second paper, "Currents Carried by Sodium and Potassium Ions Through the Membrane of the Giant Axon of Loligo", described the measured changes in the action potential using different sodium concentrations, and how the total ionic current was distributed into sodium and potassium currents [11]. The third paper, "The Components of Membrane Conductance in the Giant Axon of Loligo", investigated the effect of sudden potential changes on the action potential [12]. The fourth paper, "The Dual Effect of Membrane Potential on Sodium Conductance in the Giant Axon of Loligo", deals with the 'inactivation' process as sodium permeability is returned to normal during the transition between the depolarization and repolarization phases during an action potential [10]. Finally, the fifth and final paper in the series, "A Quantitative Description of Membrane Current and Its Application to Conduction and Excitation in

Nerve", reviews the first four papers, and then describes the mathematical model developed based on the research [9].

4.2 Mathematical Description

Hodgkin-Huxley theory explains the relationship, within a patch of axonal membrane, between ionic currents (sodium, potassium and leakage), capacitive currents and the membrane current density [9-13]. This relationship is defined by (4.1).

$$J_m = J_c + J_{Na} + J_K + J_L \tag{4.1}$$

where J_m, J_c, J_{Na}, J_K and J_L are the membrane, capacitance, sodium, potassium and leakage current densities. The equation can be rewritten using Ohm's Law and the constitutive law for the membrane capacitance as:

$$J_m = C_m \frac{\partial V_m}{\partial t} + g_K(V_m - V_k) + g_{Na}(V_m - V_{Na}) + g_L(V_m - V_L) \tag{4.2}$$

where C_m is the transmembrane capacitance; g_{Na}, g_K and g_L are the conductances for sodium, potassium and leakage through the membrane; V_{Na}, V_K and V_L are the sodium, potassium, and leakage equilibrium potentials. The sodium and potassium potentials are defined by the Nernst equilibrium equations:

$$V_{Na} = \left(\frac{RT}{F}\right)\log\left(\frac{C^o_{Na}}{C^i_{Na}}\right) \qquad V_K = \left(\frac{RT}{F}\right)\log\left(\frac{C^o_k}{C^i_k}\right) \tag{4.3}$$

V_{Ion} is the Nernst potential for the specific ion; R is the molar gas constant; T is absolute temperature and F is Faraday's constant. The variables C^o_{Ion} and C^i_{Ion} are the extracellular and intracellular ionic concentrations respectively. The leakage equilibrium is a constant, -49.0E-3 mV. The model equates a neuron with an equivalent circuit model (Figure 4.1).

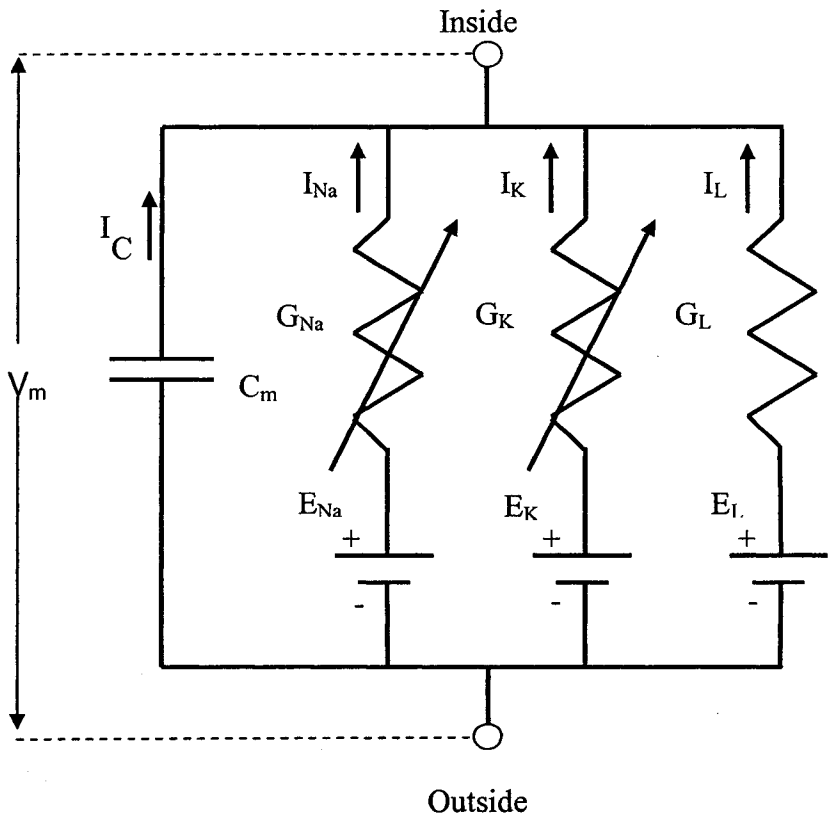


Figure 4.1. The Hodgkin-Huxley equivalent circuit diagram depicting a patch of a neuronal cell membrane in terms of ionic conductances (G), current densities (I), ionic potentials (E) and membrane capacitance s (C_m).

The cell membrane modeled by this circuit is dependant upon the ionic currents due to sodium and potassium. All other currents are categorized as "leakage" or I_L. Nernst equilibrium potentials for sodium and potassium are V_{Na} and V_K respectively, and the constant leakage potential is shown as V_L. Finally, the conductances for sodium, potassium and leakage with respect to time and membrane voltage are g_{Na}, g_K and g_L. These sodium and potassium conductances are defined in 4.4.

$$g_{Na}(V_m,t) = \bar{g}_{Na} m^3(V_m,t) h(V_m,t) \qquad g_K(V_m,t) = \bar{g}_K n^4(V_m,t) \qquad 4.4$$

where $\bar{g}_{Na}$ and $\bar{g}_{k}$ are maximal sodium and potassium conductances and m, h, and n are variables that describe the m, h and n gating dynamics. The leakage conductance is a constant 0.3E-3 S/cm^2. The corresponding rate equations for the gating variables have the form

$$\frac{dm}{dt} = \alpha_m(1-m) - \beta_m m \qquad \frac{dh}{dt} = \alpha_h(1-h) - \beta_h h) \qquad \frac{dn}{dt} = \alpha_n(1-n) - \beta_n n \qquad 4.5$$

The α and β parameters in the above equations are the rate constants which define the rate at which particles move from the inside of the cell membrane to the outside. These equations define the net rate of change in m, h and n by relating the gating particles that are moving between the inside to outside of the cell membrane to the gating particles that are moving between the outside to the inside of the cell membrane [20]. Hodgkin and Huxley fit the rate constants to analytical expressions dependent on the membrane potential [9]. There are numerous versions of these equations in the literature, and the equations actually used in the project can be found in Weiss [18]. The rate constant equations are

$$\alpha_m = \frac{-0.1(V_m+35)}{e^{-0.1(V_m+35)}-1} \qquad \beta_m = 4e^{-(V_m+60)/18}$$

$$\alpha_h = 0.07e^{-0.05(V_m+60)} \qquad \beta_h = \frac{1}{1+e^{-0.1(V_m+30)}}$$

$$\alpha_n = \frac{-0.01(V_m+50)}{e^{-0.1(V_m+50)}-1} \qquad \beta_n = 0.125e^{-0.0125(V_m+60)}$$

The m, h, and n differential equations can also be developed in terms of a time constant (τ_x) and a final value(x_∞). The individual equations for each gate using time constants and the final values are

$$\frac{dm}{dt} = \frac{m_\infty - m}{\tau_m} \qquad \frac{dh}{dt} = \frac{h_\infty - h}{\tau_h} \qquad \frac{dn}{dt} = \frac{n_\infty - n}{\tau_n}$$

The time constants are defined in terms of the rate constants described above. The three time constant equations are

$$\tau_m = \frac{1}{\alpha_m + \beta_m} \qquad \tau_m = \frac{1}{\alpha_m + \beta_m} \qquad \tau_m = \frac{1}{\alpha_m + \beta_m}$$

The final values used are also defined using the rate constants. The three final value equations are

$$m_\infty = \frac{\alpha_m}{\alpha_m + \beta_m} \qquad h_\infty = \frac{\alpha_h}{\alpha_h + \beta_h} \qquad n_\infty = \frac{\alpha_n}{\alpha_n + \beta_n}$$

These equations along with the numerical parameters found in table 4.1 make up the generalized Hodgkin-Huxley model for a patch of axonal membrane [18].

Table 4.1. Numerical parameters used in the Hodgkin-Huxley model.

Element		Value	Units
Membrane capacitance per unit area	Cm	1.0e-6	F/cm^2
Intracellular sodium ion concentration	C_{Na}^{i}	50.0E-3	mol/L
Extracellular sodium ion concentration	C_{Na}^{O}	491.0E-3	mol/L
Intracellular potassium ion concentration	C_{k}^{i}	400.0E-3	mol/L
Extracellular potassium ion concentration	C_{k}^{O}	20.11E-3	mol/L
Maximum sodium conductance per unit area	$\bar{g}_{Na}$	120.0E-3	S/cm^2
Maximum potassium conductance per unit area	$\bar{g}_{K}$	36.0E-3	S/cm^2
Leakage conductance per unit area	g_L	0.3E-3	S/cm^2
Leakage potential	V_L	-49.0E-3	mV

4.3 Specifics of Electrical Excitability in the Hodgkin-Huxley Model

In order to validate the code model developed for this project, it is necessary to discuss some specifics of the Hodgkin-Huxley model. The details of the model will be developed here, while the comparison and validation will occur in section 9.

4.3.1 m, h, & n values

The first detail involves the actual values of the gating variables, m, h, and n at rest as well as throughout the generation of an action potential. As discussed in section 3, these gating variables tie to the m, h, and n gates which react to depolarization of the cellular membrane at the initiation of an action potential. The "m" sodium activation gate reacts quickly, opening once a depolarization of 10-20 mV occurs. The "h" activation gate begins to close upon depolarization; however, it closes more slowly compared to the "m" gate activity. The "h" gate remains closed until the refractory period has passed Finally, the potassium "n" gate begins to open once sufficient

depolarization has occurred to start the action potential, but like the sodium "h" gate, it is also slow to react. It remains open past the point of repolarization to the resting potential of the cellular membrane. This phenomenon is illustrated on page 202, figure 4.32 in Weiss [18] and will be used as a reference during the validation of the code model in section 8.

4.3.2 Repetitive Activity

Studies have shown that a suprathreshold current applied to a nerve cell over time will cause multiple action potentials [22]. When currents slightly above the cells threshold are used, only a single action potential is created. When higher continuous current amplitudes are applied, multiple action potentials are seen with the latter action potentials maximum amplitude being less than the initial action potential. Ultimately, subsequent action potentials are blocked demonstrating a property called a depolarization block. This effect is caused by the continuous current's ability to decrease h and increase n over time. These changes affect the sodium and potassium conductances which in turn reduces the action potential amplitude ultimately resulting in the block. This affect is called repetitive activity, and the Hodgkin-Huxley model exhibits the basics of this phenomenon. An illustration for different stimulus amplitudes can be found on page 232, figure 4.60 in Weiss [18], and will be used as a reference during the validation of the code model in section 8.

4.3.3 Accommodation

If a current stimulus is applied to a nerve cell below the threshold value for that cell, and then slowly increased over time, no action potential is generated even though the

threshold value for the cell is subsequently exceeded. Known as accommodation, this phenomenon has been studied since the mid nineteenth century [23]. Later studies using the space clamped squid axon apply ramped currents of differing slopes [22;24]. These studies found slope values below which no action potential was generated despite the threshold value of current being exceeded under normal step conditions. The Hodgkin-Huxley model does not exhibit accommodation at the standard parameters of the model; however, if the maximum sodium conductance ($\bar{g}_{Na}$) is lowered to 80.0E-3 S/cm^2, the Hodgkin-Huxley model does exhibit the accommodation phenomenon [18]. The accommodation effect is shown on page 228, figure 4.56 in Weiss [18], and will be used as a reference during the validation of the code model in section 8.

4.3.4 Anode-Break Excitation

Another phenomenon present in nerve cells happens when a hyperpolarizing current is passed into a nerve cell for a period of time long enough for the membrane potential and all gating variables to stabilize at their steady-state values. When the current is subsequently removed from the cell, an action potential is generated. This is known as a anode-break excitation, and it occurs due to the increase in h and the decrease in n which affects the potassium conductance. This lowers the threshold necessary for initiating an action potential. This new threshold is achieved when the hyperpolarized current is removed, and the cell is repolarized to its normal resting potential [18]. The calculated anode-break excitation phenomenon is shown on page 230, figure 4.59 [18], and will be used as a reference during the validation of the code model in section 8.

4.3.5 Subthreshold Oscillations

When a continuous pulse of a relatively small subthreshold current is passed into a nerve cell, the membrane potential exhibits highly damped oscillations [9;25]. This effect happens for both positive and negative currents. Subthreshold oscillations are seen in the Hodgkin-Huxley model. The calculated subthreshold oscillations phenomenon is shown on page 233, Figure 4.61of Weiss [18], and will be used as a reference during the validation of the code model in section 8.

4.3.6 Effect of Temperature

There have been many papers written in the literature regarding the effects of temperature on the firing of an action potential in a neuron. Hodgkin and Huxley documented temperature dependence in 1952 [26]. This work was the culmination of various experiments completed on the squid giant axon which defined the Hodgkin-Huxley model that this dissertation research is based on. However, in this paper, the only temperature affect was the scaling of the rate equations (m, h, and n) by a Q_{10} factor. Huxley followed this work with a paper in 1959 [27] which adds the temperature effects on conductances shown in the Nernst equilibrium potentials to the α and β scaling effects. Since these papers, there have been numerous papers published on the effects of temperature on myelinated and nonmyelinated neurons [28-32]. These papers provide documentation for a variety of Q_{10} values used.

As mentioned, temperature dependence in the Hodgkin-Huxley model is incorporated in two ways [9]. The temperature dependence is included within the Nernst equilibrium potential equations seen in equation 2.1. The Nernst potential of the ions involved is proportional to absolute temperature. Temperature also has an impact on the

Hodgkin-Huxley model through the rate constants. The m, h, and n differential equations are scaled by a constant temperature factor, K_t [18]. This leads to the following equations based on the time constant and final values.

$$\frac{dm}{dt}=\left(\frac{m_\infty - m}{\tau_m}\right)K_t \qquad \frac{dh}{dt}=\left(\frac{h_\infty - h}{\tau_h}\right)K_t \qquad \frac{dn}{dt}=\left(\frac{n_\infty - n}{\tau_n}\right)K_t \tag{4.6}$$

where K_t is the temperature coefficient (Q_{10}) value raised to the power representing the increase in the rate constant for every 10°C change in the temperature at which the kinetics were measured as per (4.7).

$$K_T = Q_{10}^{\frac{(T_c - 6.3)}{10}} \tag{4.7}$$

The T_c value is the measured temperature. A Q_{10} value of 3 is documented for the squid giant axon [9].

CHAPTER 5

THE SPICE CIRCUIT SIMULATOR

With the birth of integrated circuits in 1961, a new problem in the electronic industry was encountered. Unlike the board-level designs before them, integrated circuits are unable to be breadboarded before the manufacturing process begins. Because of the high costs of the photolithography process used to manufacture integrated circuits, a new tool needed to be developed to test designs in order to ensure the design acted as intended once manufactured in mass. The SPICE (Simulation Program with Integrated Circuits Emphasis) simulation program was created for this purpose in 1975. Today, simulating an integrated circuit with SPICE is the industry-standard way to verify circuit operation before beginning the manufacturing process.

SPICE in its many forms utilizes a text file called a "netlist" also known as a "deck" that is run by the simulator. This netlist file or deck describes the circuit elements (transistors, resistors, capacitors, etc.) and their connections. These elements and connections are then translated into equations to be solved. These nonlinear differential equations are then solved using implicit integration methods, Newton's method and sparse matrix techniques.

5.1 Origins of SPICE

SPICE was created in 1972 by Larry Nagel and Donald Pederson at the Electronics Research Laboratory of the University of California, Berkeley. The first two versions were coded in Fortran which ran on mainframe computers. Later versions have been coded in C; however, the circuit description files or netlists still use a Fortran-like syntax.

SPICE was largely a derivative of the CANCER simulation program created by Nagel and Rohrer in 1971 [33]. CANCER was an acronym for "Computer Analysis of Nonlinear Circuits, Excluding Radiation". The name was chosen due to the requirement placed on many circuit simulators developed during the 1960s by United States Department of Defense contracts. These contracts required simulators built for the government to provide the ability to evaluate a circuit's radiation hardness.

Initial versions of SPICE used nodal analysis. Because of this approach, ideal voltage sources and inductors could not be included in the circuit. Since then, a modified nodal analysis technique utilizing different algorithms to translate all circuit analysis problems into a single problem or multiple simpler problems is used. This technique allows for the problem to be solved by solving a linear simultaneous equation (i.e. non-linear circuits are solved using a Newton-Raphson algorithm, which linearizes non-linear elements in a circuit). Transient analysis is performed using the trapezoid or Gear integration algorithm.

Free versions of SPICE are available for most computing platforms. Versions such as XSPICE, NGSPICE, TCLSPICE, and SpiceOpus Light provide free simulators that are readily available for use in the academic environment. Two of these versions,

NGSPICE (both the Windows and Linux versions) and SpiceOpus, were used extensively for this project. Of fundamental importance to this work is the functionality developed by the XSPICE implementation which allows the development of code models.

5.2 XSPICE

XSPICE is an enhanced SPICE simulation engine, developed by Georgia Tech Research Institute for the United States Air Force as part of the Automatic Test Equipment Software Support Environment (ATESSE), version 2.0. XSPICE was designed to assist engineers with the development of software for the control of Automatic Test Equipment (ATE). It significantly added to the functionality of SPICE by including a number of abstract simulation models as well as mixed analog/digital circuit simulation. For the work described in this dissertation, the most important functional addition made by XSPICE is its code model tool kit. This toolkit provides users the ability to design and program arbitrarily complex functions instead of requiring users to only use the slower, more traditional macro-modeling approach [16]. The code model toolkit allows the user to write, compile and link newly created "electronic" models with the existing SPICE simulator. Since XSPICE's development, both free and commercial versions of SPICE have incorporated the code modeling capability developed by XSPICE. Figure 5.1 describes how the existing model library, and the user defined models, interacts with the SPICE software core.

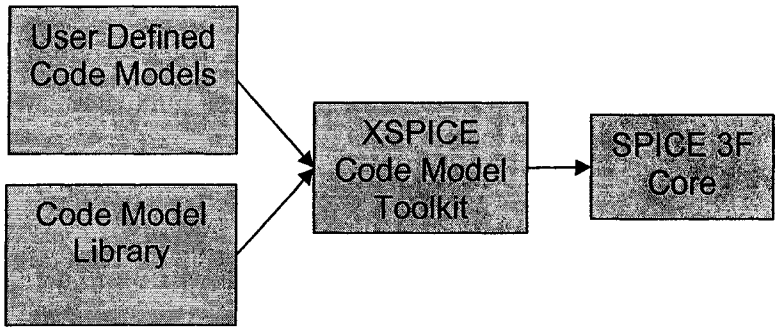


Figure 5.1. Graphical Representation of XSPICE Code Model toolkit implementation in SPICE.

5.3 NGSPICE

Development began on NGSPICE in 1999 by a small group of internet users based on SPICE 3F5 and it is licensed under the standard BSD license. Versions are available for both LINUX and Windows platforms. The Windows version requires a Linux emulator, and runs much like the Linux version. The simulator includes an interpreted programming language called Nutmeg, which allows interactive Spice sessions. XSPICE functionality is included with full support of the code model toolkit. Documentation and software downloads can be found at http://ngspice.sourceforge.net/index.html [34]. A screenshot of the LINUX emulator, NGSPICE user interface, and its output using its graphing capability is shown in figure 5.2.

5.4 SPICE OPUS

First released in 1999, SpiceOpus is a circuit simulator with optimization utilities. It is based on the original Berkeley source code and is available for the Windows and Linux operating systems. Unlike the Windows version of NGSPICE, SpiceOpus is a resident application in the Windows environment, and does not need a Linux emulator to run. XSPICE functionality is included with full support of the code model toolkit. The simulator includes an interpreted programming language called Nutmeg, which allows

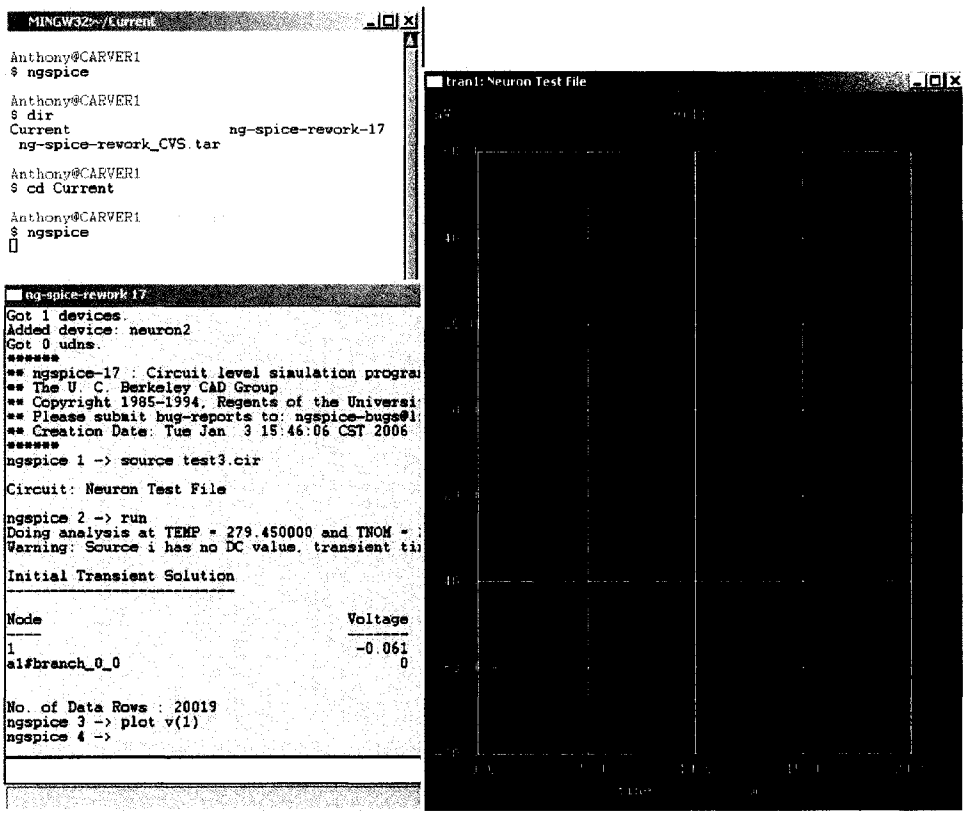


Figure 5.2. Screenshot NGSPICE user interface and its graphical output.

interactive Spice sessions. A screenshot of the SPICE OPUS user interface and its output using its graphing capability is shown in Figure 5.3

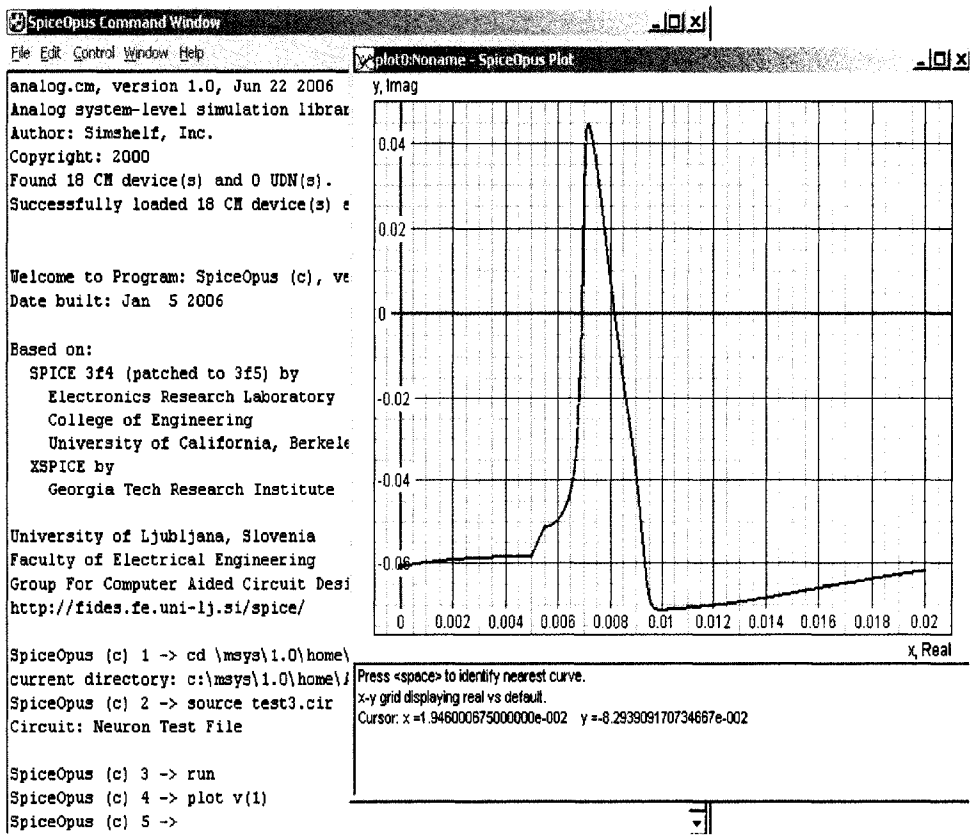


Figure 5.3. Screenshot SPICE OPUS user interface and its graphical output.

CHAPTER 6

CODE MODEL SUBSYSTEM

The code model system developed in XSPICE and implemented into many current day circuit simulators gives a user the ability to define arbitrarily complex systems without having to exclusively use the predefined set of discrete components within the SPICE simulator. The code model functionality is utilized, on the user end, by creating and editing the InterFace Specification (IFS) file and the MOdel Definition (MOD) file for each model [35]. These files work together to define the data structures used within the model to transfer data to and from the main SPICE simulator as well as the detailed model system characteristics.

6.1 Interface Specification File

The InterFace Specification file (IFS) is a text file containing naming information for the code model. It also defines the input and output ports used in simulation runs as well as any expected parameters and variables used to pass data between the code model and main SPICE simulation environment. The IFS file is broken down into 4 sections based on these four areas. The sections are named NAME_TABLE, PORT_TABLE, PARAMETER_TABLE, and STATIC_VAR_TABLE respectively and are described below.

6.1.1 Name Table

The name table section of the IFS file contains three sections. The first section defines the C function name used in the "cfunc.mod" file. Section two defines the actual name used in the .model statement within the SPICE simulation file also known as a "deck" in reference to the punch cards used to run SPICE programs in its early days of use. Finally, section three provides a description of the model being developed. The Name Table section is started by the "Name_Table:" entry followed by the individual sections table names.

6.1.1.1 C Function Name

The C Function name section defines the name of the code model and must be a valid C identifier. The table entry is started by the "C_Function_Name:" entry followed by the actual name. This name also must correspond with the function within the Model Definition file. If the two names are different, an error will result in the linking step of the code model compiling and linking procedure at the end of the code model development.

6.1.1.2 SPICE Model Name

The SPICE Model Name section defines the name used in the actual SPICE simulation deck with the ".model" line. This name can be any name the user wishes as long as it is a valid SPICE identifier, and is not restricted to the C function name. The table entry is started by the "SPICE_Model_Name:" entry followed by any valid SPICE name.

6.1.1.3 Description

The third section of the name table in the IFS file provides the user a way of commenting the general purpose of the code model. The table entry is started by the "Description:" entry followed by any C string literal.

6.1.2 Port Table

XSPICE uses ports to pass data between the code model and the SPICE simulation deck. These ports can be input, output, or a combination of in/out ports as defined by the user. The Port Table defines these ports and is broken into eight sections. This table defines input and output ports as needed by the code model. The eight sections of the Port Table are Port Name, Description, Direction, Default Type, Allowed Types, Vector, Vector Bounds, and Null Allowed. The Port Table section is started by the "Port_Table:" entry followed by the individual sections table names.

6.1.2.1 Port Name

The Port Name table entry names the ports to be used in the code model. The name must be a valid SPICE identifier and is started by the "Port_Name:" entry followed by the valid port name.

6.1.2.2 Description

The Description table entry gives the user the ability to provide a brief description of each port used in the code model. The description section is started by the "Description:" entry followed by any C string literal.

6.1.2.3 Direction

The Direction table entry specifies which direction data flows through the port used in the code model. Valid port directions include two one-way options, "in" and "out" and the bidirectional option "inout". The direction section is started by the "Direction:" entry followed by one of the three options above.

6.1.2.4 Default Type

The Default Type table entry specifies the default type of port used in the code model. Table 6.1 lists the allowed port types. The Default Type section is started by the "Default_Type:" entry followed by one of the allowed port types.

Table 6.1. Available port types and directions associated with port type within the XSPICE Code Model Toolkit.

Default Port Types		
Type	Description	Direction
D	Digital	in or out
G	Conductance (VCCS)	inout
Gd	Differential Conductance (VCCS)	inout
H	Resistance (CCVS)	inout
Hd	Differential Resistance (CCVS)	inout
I	Current	in or out
Id	Differential Current	in or out
V	Voltage	in or out
Vd	Differential Voltage	in or out
<identifier>	User Defined Type	in or out

6.1.2.5 Allowed Types

While ports have an associated default port type, other port types can be used as specified in the Allowed Types section. The allowed types must be an allowed type from

table 4.1. The Allowed Type section is started by the "Allowed_Types:" entry followed by a single allowed port type in square brackets (e.g. "[v]") or a comma or space separated list in square brackets (e.g. "[v, i, id]").

6.1.2.6 Vector

The Vector table entry allows the user to assign a vector for input or output based on the port type. The Vector section is started by the "Vector:" entry followed by the Boolean value "YES" or "TRUE" if the port is a vector or "NO" or "FALSE" if the port is not a vector. If a port is defined as a vector, it must have a valid vector bounds table entry. Otherwise, the vector bounds entry can be omitted using a "-" value in the vector bounds entry.

6.1.2.7 Vector Bounds

If a "YES" or "TRUE" value is given in the Vector table entry, an associated Vector Bounds entry must follow. This entry in the Vector Bounds field sets the size of the vector to be used. Both an upper and lower bound is given for the vector size. The Vector Bounds section is started by the "Vector_Bounds:" entry followed by a space or comma separated list in square brackets defining the lower and upper bounds of the vector (e.g. "[1 10]" or "[4, 20]").

6.1.2.8 Null Allowed

Should the user require a port remain disconnected from the circuit being simulated, the Null Allowed entry allows for this circumstance. The Null Allowed section is started by the "Null_Allowed:" entry followed by the "YES" or "TRUE" entry

if a null is allowed or a "NO" or "FALSE" entry if the port must be connected which would generate an error at runtime if the port was not connected.

6.1.3 Parameter Table

XSPICE incorporates the ability to parameterize the code model allowing for changes to system variables between simulations without having to recompile and link the code model to the SPICE system. These parameters are defined in the Parameter Table and its five sections: Parameter Name, Description, Data Type, Null Allowed, and Default Value. The Parameter Table section is started by the "Parameter_Table:" entry followed by the individual section table names.

6.1.3.1 Parameter Name

The Parameter Name section defines the individual parameter names used by the code model. The table entry is started by the "Parameter_Name:" entry followed by the actual name. The name must be a valid SPICE identifier used on the ".model" card in the simulation file.

6.1.3.2 Description

The Description table entry gives the user the ability to provide a brief description of each parameter used in the code model. The description section is started by the "Description:" entry followed by any C string literal.

6.1.3.3 Data Type

As in all programming languages, the XSPICE Code Model Toolkit requires parameters be defined by type. Available data types include Boolean, int, real, string,

and complex. The Data Type section is started by the "Data_Type:" entry followed by the valid parameter data type.

6.1.3.4 Null Allowed

The Null Allowed section of the Parameter Table allows a user to either require or not require a starting value for the given parameter at run time on the ".model" line. If the Null Allowed entry is set to "TRUE" or "YES", a value is not required, and the default value is used (see Default Value section below). If the Null Allowed value is set to "FALSE" or "NO", an error is generated if no value is specified in the ".model" line in the SPICE deck. The Null Allow section is started by the "Null_Allowed:" entry followed by the valid Boolean entry.

6.1.3.5 Default Value

If the Null Allowed section is set to "TRUE" or "YES", the Default Value section of the Parameter Table defines the default value to be used by the SPICE simulator if no value is given on the ".model" line. The Default Value section is started by the "Default_Value:" entry followed by the actual default parameter value. The default value must be of the same data type as the Data Type defined above.

6.1.4 Static Variable Table

XSPICE provides the ability to pass variables between successive iterations of the code model in the simulation environment. This is accomplished with the use of static variables (variables that are not cleared after a single iteration on the code model). These static variables are defined in the Static Variable Table and its three sections: Name,

Description, and Data Type. The Parameter Table section is started by the "Static_Var_Table:" entry followed by the individual section table names.

6.1.4.1 Name

The Name section defines the individual static variable names used by the code model. The table entry is started by the "Static_Var_Name:" entry followed by the actual name. The name must be a valid C identifier used on the ".model" card in the simulation file.

6.1.4.2 Description

The Description table entry gives the user the ability to provide a brief description of each static variable used in the code model. The description section is started by the "Description:" entry followed by any C string literal.

6.1.4.3 Data Type

As in all programming languages, the XSPICE Code Model Toolkit requires static variables be defined by type. Available data types include Boolean, int, real, string, pointer, and complex. The Data Type section is started by the "Data_Type:" entry followed by the valid static variable data type.

6.2 Model Definition File

The second file needed to create a XSPICE code model is the MOdel Definition file or MOD file. The MOD file is given the standard name "cfunc.mod" for all code models. This file contains the C programming language source code which implements

the functionality of the code model within the SPICE simulation environment. By using the previously mentioned parameters and static variables, the code model is able to work over multiple iterations of itself within the SPICE simulation environment. However, this data handling requires the use of special accessor macros to work on specific input, output, and simulator specific variables and parameters.

6.2.1 Accessor Macros

Accessor Macros provide the MOD file a capability of working with or "accessing" parameters and static variables defined in the IFS file. The accessor macros are broken into several distinct types to include Circuit Data, Parameter Data, Input Data, Output Data, and Static Variable macros. These accessor macros are accessible within the MOD file. Table 6.2 lists the complete set of accessor macros available in XSPICE [35]. The remainder of this section is devoted to the accessor macros used in this code model implementation.

6.2.1.1 Circuit Data

The circuit data group of accessor macros provides the code model programmer with information concerning the state of and conditions within the code model throughout multiple iterations of the code model during the simulation run. The specific circuit data accessor macros used in this code model implementation included ARGS, INIT, T(n), and TEMPERATURE.

6.2.1.1.1 "ARGS"

ARGS is the macro passed in the argument list of the C function within the MOD file. It provides a means of referencing each model to the rest of the macro values. It is the only argument allowed in the argument list, and it is required.

6.2.1.1.2 "INIT"

INIT is basically a binary test to find out whether or not the code model has been called before its current instance. INIT is set to "1" if this is the first call to the code model instance, and it is set to "0" if it is not.

6.2.1.1.3 "T(n)"

T(n) is a double (C numeric data type) vector giving the code model programmer insight into the timing during a transient analysis. The vector contains the current time at T(0), and the last accepted timepoint at T(1). This also gives the programmer a way of knowing the current timestep by subtracting T(1) from T(0).

6.2.1.1.4 "TEMPERATURE"

TEMPERATURE is a double (C numeric data type) holding the value of the temperature for the current simulation run. This temperature is set for the SPICE simulation run in the SPICE deck using the temp="value" and tnom="value" in the .options line. This allows the code model programmer the ability to integrate the simulation temperature easily with the code model functionality where temperature dependence is required.

Table 6.2. Accessor Macros used within the XSPICE Code Model Toolkit.

Accessor Macros Used in the XSPICE Code Model Toolkit			
Name	Type	Args	Description
ARGS	Mif_Private_t	<none>	Standard argument to all code model functions
CALL_TYPE	enum	<none>	Type of model evaluation call: ANALOG or EVENT
INIT	Boolean	<none>	Is this the first call to the code model
ANALYSIS	enum	<none>	Type of analysis: DC, AC, TRANSIENT
FIRST_TIMEPOINT	int	<none>	Integer that takes the value of 1 or 0 on whether this is the first call to the code modes instance or not , respectively
TIME	Double	<none>	Current analysis time (same as T(0))
T(n)	int	index	Current and previous analysis times (T(0) = TIME = current analysis time, T(1) = previous analysis time)
RAD_FREQ	double	<none>	Current analysis frequency in radians per second
TEMPERATURE	double	<none>	Current analysis temperature
PARAM	Context Dependent	name[i]	Value of the parameter
PARAM_SIZE	int	name	Size of parameter vector
PARAM_NULL	Boolean_t	name[i]	Was the parameter not included on the SOICE .model card?
PORT_SIZE	int	name	Size of port vector
PORT_NULL	Mif_Boolean_t	name	Has this port been specified as unconnected?
LOAD	double	name[i]	The digital load value placed on a port by this model
TOTAL_LOAD	double	name[i]	The total of all loads on the node attached to this event-driven port
INPUT	double or void *	name[i]	Value of analog input port
INPUT_STATE	enum	name[i]	State of a digital input: ZERO, ONE, or UNKNOWN
INPUT_STRENGTH	enum	name[i]	Strength on digital input: STRONG, RESISTIVE, HI_IMPEDENCE, or UNDETERMINED
OUTPUT	double or void *	name[i]	Value of analog output port
OUTPUT_CHANGED	Boolean_t	name[i]	Has a new value been assigned to this event-driven output by the model
OUTPUT_DELAY	double	name[i]	Delay in seconds for an event-driven output
OUTPUT_STATE	enum	name[i]	State of a digital output: ZERO, ONE, or UNKNOWN
OUTPUT_STRENGTH	enum	name[i]	Strength on digital output: STRONG, RESISTIVE, HI_IMPEDENCE, or UNDETERMINED
PARTIAL	double	y[i],x[i]	Partial derivative of output y with respect to input x
AC_GAIN	Complex_t	y[i],x[i]	AC gain of output y with respect to input x
STATIC_VAR	Context Dependent	name	Value of static variable

6.2.1.2 Parameter Data, Input, and Output Data, and Static Variable

While the circuit data group has accessor macros which allow the programmer to monitor conditions within the code model during the simulation, it is the parameter data, input and output data, and the static variable accessor macros that truly make the code model function. These accessor macros allow the code model programmer to set parameters within the IFS file, pass data between the code model instance and the SPICE simulation environment, as well as maintain variables between multiple iterations of code model itself.

6.2.1.2.1 PARAM(parameter_ name)

The PARAM accessor macro allows the code model programmer to access the parameters set in the IFS file mentioned earlier. These parameters can pass default values into the simulation allowing the programmer to set and change initial conditions without having to recompile and link the code model. The following example loads the parameter "v_rest" defined in the IFS file into the C variable "Voltage_rest".

Voltage_rest = PARAM(v_rest);

6.2.1.2.2 INPUT(port_name)

The INPUT accessor macro provides a path for the code model programmer to access any input ports defined in the IFS file. This allows input from the SPICE circuit simulation to be acted on by the code model's functionality. The following example saves data from the input port "a" into the C variable "I_in".

I_in = INPUT(a);

6.2.1.2.3 OUTPUT(port_name)

The OUTPUT accessor macro provides a path for the code model programmer to send data back to the SPICE simulation from the code model via any output ports defined in the IFS file. This allows output from the code model to be acted on by the SPICE circuit simulation. The following example saves data from the OUTPUT port "b" into the C variable "I_in".

OUTPUT(b) = Vm;

6.2.1.2.4 STATIC_VAR(stativ_var_name)

The STATIC_VAR accessor macro allows the code model programmer to access the static variables set in the IFS file mentioned earlier. These static variables act as nonvolatile memory to save data through out the many iterations of the code model during a SPICE simulation. The following example allocates static memory as required by the C programming language. The example then loads the static variable "previous_m" defined in the IFS file and possibly used in the previous iteration of the code model into the C variable "previous_m".

STATIC_VAR(previous_m) = (double *) malloc(sizeof(double));

previous_m = STATIC_VAR(previous_m);

6.3 Device Model Creation and Setup in SPICE

Both NGSPICE and SPICE OPUS are well documented on creating the device models and building them once created. Procedures for both applications are similar with the main difference being an additional step required by SPICE OPUS at the end of the

process. Since SPICE OPUS is a Windows application, it requires, as a final step, compilation of all files into a multithreaded dynamically linked library (.dll) file.

6.3.1 NGSPICE Device Model Creation

NGSPICE was developed in 1999 by a small group of internet users based on SPICE 3F5 and it is licensed under the standard BSD license. Versions used for this dissertation include both LINUX and Windows versions. The Windows version requires a Linux emulator, and runs much like the Linux version.

The procedures needed to create, compile and link the newly developed device model can be broken down into three main steps. Step 1 involves setting up the directory structure required for the device model. The code model tool kit present in XSPICE requires the user to place the device model files in the "ng-spice\src\icm" folder. The remainder of the steps will be described as completed for the neuron device model. Should problems arise, reviewing the files and folders used by other device models present in the inventory is a helpful option. A new folder was created for each individual device model. In the case of the device model for this dissertation, the folder "neuron" was created. Inside the neuron folder, an icm_neuron folder was created. This folder will hold the "ifspec.ifs" and "cfunc.mod" files used to define the data structures and device model functionality as described in Paras. 6.1 and 6.2. In the "neuron" folder, a new file was created called modpath.lst which contains a list of user devices. In this case, "icm_neuron" is the only entry. The next step was to edit the "src/xspice/icm/makedefs.in" and alter the CMDIRS line to include the "neuron" directory. The last step was to run the command "make" in the "src\xspice\icm" directory. This runs the "make" file which contains the information for compiling and

linking needed to connect the device model to the SPICE engine. Once completed, the device model could be found in "src/xspice/icm/neuron/neuron.cm". Further, more detailed information, can be found on the NGSPICE website and inside the "src\xspice" folder under the "README" file [34].

6.3.2 SPICE OPUS Device Model Creation

First released in 1999, SpiceOpus is a circuit simulator with optimization utilities. It is based on the original Berkeley source code and is available for the Windows and Linux operating systems. Unlike the Windows version of NGSPICE, SPICE OPUS is a resident application in the Windows environment, and does not need a Linux emulator to run. XSPICE functionality is included with full support of the code model toolkit.

The procedures needed to create, compile and link the newly developed device model in SPICE OPUS is very similar to the steps used for NGSPICE. One main difference between the two versions is everything compiled must be compiled for use in a multithread dll library. The first step is to create one directory for each device model. All files created will be placed into this directory. Next, the "ifspec.ifs" and "cfunc.mod" files are created. Next, compile the "ifspec.ifs" file by using the cmpp utility with the command "cmpp –ifs". This results in a "ifspec.c file". Now, compile the cfunc.mod file using the cmpp utility with the command "cmpp –mod". This results in a "cfunc.c" file. The next step is to compile both .c files using a C compiler. It is important to note that the files must be compiled as a multithreaded .dll program. The include files for compilation are in the xsource/include directory in the SPICE OPUS tree. Now there are two ".obj" files that will be linked later with other ".obj" files to produce the dll (.cm) library. To build the library, list all CM directories in the "cmpath.lst" file. Next, compile the ".lst"

files using the command "cmpp –lst". This step will create two files: "cmextern.h", "cminfo.c". The "dlinfo.h" file must now be edited to enter the information about the ".cm" library. The next step is to compile the "dlmain.c" with multithreaded dll options set. By defining the macro CM_WINDOWS, you specify that this is a Windows compile. Finally, link together all CM ".obj" files with the "dlmain.obj" file to obtain a multithreaded .dll., and change the files extension to ".cm"

CHAPTER 7

DEVICE MODEL DESCRIPTION

As detailed in chapter seven, the XSPICE code model toolkit requires two files to define the device model created by the user. The two files required are the "ifspec.ifs" and "cfunc.mod" files. The "ifspec.ifs" file defines the data structures used to pass data to, from, and within the device model during a given simulation run, as well as defining parameters for use with the device model. The "cfunc.mod" file defines the behavior of the device model using the C programming language. The actual files for the device model detailed in this dissertation are broken into sections in 7.1 and 7.2 of this chapter. This is done to discuss the contents of the files as well as document the specifics of the files and how they define the functionality of the Hodgkin-Huxley based device model. Only pertinent areas will be covered in the description.

7.1 Interface Specification File

The "ifspec.ifs" file is broken into five sections for the source code description. Section 1 is the basic file information. Section 2 contains the C language include and variable declarations used in the device model. Section 3 is the beginning of the first call to the device model by SPICE. It loads the parameters set in the .model line of the netlist

file if present. Any parameter not set in the ".model" line will be loaded with the default value set in the ifspec.ifs file or calculated as required. Section 4 allocates static memory locations for the static variables defined in the "ifspec.ifs" file. Section 5 sets the initial conditions of the device model and the static variables associated with the device model. Section 6 loads the static variable values for the model values from the last time step as well as calculates the current values of the alpha and beta values for the current time step. Finally, section 7 calculates the equations of the Hodgkin-Huxley model culminating in the membrane potential for the current time step.

7.1.1 "ifspec.ifs" File - Section 1

Figure 7.1 is section one of the "ifspec.ifs" file. It is the basic descriptive element of the file common to all programs. This section details information concerning program name, author, and modifications made to the original file, as well as a summary of the files purpose.

Section 1

```
/*==============================================
Program Name - "ifspec.ifs"
AUTHOR
Anthony S. Carver, Louisiana Tech, College of Engineering and Sciences
MODIFICATIONS
None
SUMMARY
   This file contains the interface specification file for the
   Neuron code model.
==============================================*/
```

Figure 7.1. Section one of device model "ifspec.ifs" file.

7.1.2 "ifspec.ifs" File - Section 2

Figure 7.2 is section 2 of the "ifspec.ifs" file. It is the Name Table section described in 6.1.1. The name table contains three entries. The first entry defines the C function name used in the "cfunc.mod" file; in this case, the function name is "neuron". Entry two defines the actual name used in the .model statement within the SPICE simulation file also known as a "deck" in reference to the punch cards used to run SPICE programs in its early days of use. Again, for this device model, the name used in the ".model" line of the netlist file is "neuron". Finally, entry three provides a description of the model being developed.

```
Section 2

NAME_TABLE:

C_Function_Name:        neuron
Spice_Model_Name:       neuron
Description:            "Hodgkin-Huxley Code Model"
```

Figure 7.2. Section two of device model "ifspec.ifs" file.

7.1.3 "ifspec.ifs" File - Section 3

Figure 7.3 is section 3 of the "ifspec.ifs" file. It details the port connection used by the device model to transfer data to and from the SPICE simulation. The Port Table is described in 6.1.2. The port for the device model developed is named "a" and is an input and output port allowing the device model to send and receive data through the single port. The port's default type is "h" which is defined in table 6.1 as a resistance (Current Controlled Voltage Source) port. The port has no other allowed port type.

Section 3

PORT_TABLE:

```
Port_Name:      a
Description:    "Neuron Input/Output port"
Direction:      inout
Default_Type:   h           /*Current Controlled Voltage Source*/
Allowed_Types:  [h]
Vector:         no
Vector_Bounds:  -
Null_Allowed:   no
```

Figure 7.3. Section three of device model "ifspec.ifs" file.

7.1.4 "ifspec.ifs" File - Section 4

Figure 7.4 is section 4 of the "ifspec.ifs" file. It is the Parameter Table. The Parameter Table is detailed in 6.1.3. This section defines the parameters used in the ".model" line of the netlist file and sets the default values used if the parameter is not set in the ".model" line. One item to note is the value of v_rest equaling 0. This is done to allow the device to look for a value, and if the value is anything other than zero, that value is used. If the resting potential is not set in the ".model" line, the zero value is passed to the device model as the default. Within the "cfunc.mod" file, if the value passed is zero, the device model calculates the resting potential based on the other initial conditions passed to the model. The Description line provides the information concerning the 14 different parameters defined in this section.

Section 4

PARAMETER_TABLE:

Parameter_Name:	v_rest	cap
Description:	"Resting voltage"	"Capacitance Value"
Data_Type:	real	real
Default_Value:	0	1.0E-6
Limits:	-	-
Vector:	no	no
Vector_Bounds:	-	-
Null_Allowed:	yes	yes

PARAMETER_TABLE:

Parameter_Name:	ci_na	co_na
Description:	"Na intracell conc"	"Na intracell conc"
Data_Type:	real	real
Default_Value:	50.0E-3	491.0E-3
Limits:	-	-
Vector:	no	no
Vector_Bounds:	-	-
Null_Allowed:	yes	yes

PARAMETER_TABLE:

Parameter_Name:	ci_k	co_k
Description:	"K intracell conc"	"K extracell conc"
Data_Type:	real	real
Default_Value:	400.0E-3	20.11E-3
Limits:	-	-
Vector:	no	no
Vector_Bounds:	-	-
Null_Allowed:	yes	yes

PARAMETER_TABLE:

Parameter_Name:	max_gna	max_gk
Description:	"Max Sodium Conductance"	"Max Potassium Conductance"
Data_Type:	real	real
Default_Value:	120.0E-3	36.0E-3
Limits:	-	-
Vector:	no	no
Vector_Bounds:	-	-
Null_Allowed:	yes	yes

Figure 7.4. Section one of device model "ifspec.ifs" file.

PARAMETER_TABLE:		
Parameter_Name:	g_l	v_l
Description:	"Leakage conductance"	"Leakage voltage"
Data_Type:	real	real
Default_Value:	0.3E-3	-49.0E-3
Limits:	-	-
Vector:	no	no
Vector_Bounds:	-	-
Null_Allowed:	yes	yes
PARAMETER_TABLE:		
Parameter_Name:	cell_radius	cell_length
Description:	"neuron radius-meters"	"neuron length-meters"
Data_Type:	real	real
Default_Value:	1	1
Limits:	-	-
Vector:	no	no
Vector_Bounds:	-	-
Null_Allowed:	yes	yes
PARAMETER_TABLE:		
Parameter_Name:	q10	compartment_number
Description:	"temperature factor"	"1 = single compartment, 2 = more than 1"
Data_Type:	real	int
Default_Value:	3	1
Limits:	-	-
Vector:	no	no
Vector_Bounds:	-	-
Null_Allowed:	yes	yes

Figure 7.4. Continued.

7.1.5 "ifspec.ifs" File - Section 5

Figure 7.5 is section 5 of the "ifspec.ifs" file It is the Static Variable section described in 6.1.4. The Static Variable table defines the static variables, defined as pointers, which will persist through the multiple passes through the model during the entire SPICE simulation. These variables contain the previous values used to calculate the m, h, n, and current potential values using the Euler method of numerical integration.

Section 5

STATIC_VAR_TABLE:
Static_Var_Name: previous_voltage
Data_Type: pointer
Description: "iteration holding previous voltage value"

STATIC_VAR_TABLE:
Static_Var_Name: previous_m
Data_Type: pointer
Description: "iteration holding previous m value"

STATIC_VAR_TABLE:
Static_Var_Name: previous_h
Data_Type: pointer
Description: "iteration holding previous h value"

STATIC_VAR_TABLE:
Static_Var_Name: previous_n
Data_Type: pointer
Description: "iteration holding previous n value"

STATIC_VAR_TABLE:
Static_Var_Name: previous_current
Data_Type: pointer
Description: "iteration holding previous current value"

STATIC_VAR_TABLE:
Static_Var_Name: previous_voltage_temp
Data_Type: pointer
Description: "iteration holding variable for limiting"

STATIC_VAR_TABLE:
Static_Var_Name: previous_m_temp
Data_Type: pointer
Description: "iteration holding previous m value"

STATIC_VAR_TABLE:
Static_Var_Name: previous_h_temp
Data_Type: pointer
Description: "iteration holding previous h value"

STATIC_VAR_TABLE:
Static_Var_Name: previous_n_temp
Data_Type: pointer
Description: "iteration holding previous n value"

Figure 7.5. Section five of device model "ifspec.ifs" file.

```
STATIC_VAR_TABLE:
Static_Var_Name:        previous_current_temp
Data_Type:              pointer
Description:            "iteration holding previous current value"

STATIC_VAR_TABLE:
Static_Var_Name:        lastT
Data_Type:              pointer
Description:            "iteration holding previous T(0)"

STATIC_VAR_TABLE:
Static_Var_Name:        lastCurrent
Data_Type:              pointer
Description:            "Holds input current between iterations"
```

Figure 7.5. Continued.

7.2 Model Definition File

The "cfunc.mod" file is broken into seven sections for the source code description. Section 1 is the basic file information. Section 2 contains the include and variable declarations used in the device model. Section 3 is the beginning of the first call to the device model by SPICE. This section is responsible for loading the parameters set in the .model line of the netlist file, if present. Any parameter not in the .model line will be loaded with the default value set in the ifspec.ifs file or calculated as required. Section 4 allocates static memory locations for the static variables defined in the ifspec.ifs file. Section 5 sets the initial conditions of the device model and the static variables associated with the device model. Section 6 loads the static variable values for the model values from the last time step as well as calculates the α and β values for the current time step. Finally, section 7 calculates the equations of the Hodgkin-Huxley model culminating in the membrane potential for the current time step.

7.2.1 "cfunc.mod" File - Section 1

Figure 7.6 is section 1 of the "cfunc.mod" file. It is the basic descriptive elements of the file common to all programs. This section details information concerning program name, author, and modifications made to the original file, as well as a summary of what the files purpose is.

Section 1

```
/*=================================================

CFUNC.mod

AUTHOR
Anthony S. Carver, Louisiana Tech, College of Engineering and Sciences

MODIFICATIONS
None

SUMMARY
   This file contains the model-specific routines used to
   functionally describe the Neuron code model.

=================================================*/
```

Figure 7.6. Section one of device model "cfunc.mod" file.

7.2.2 "cfunc.mod" File - Section 2

Figure 7.7 is section 2 of the "cfunc.mod" file. It defines the pointers used for the static variables read in from the "ifspec.ifs" file as well as the model variables used for the Hodgkin-Huxley model calculations. This section also defines the constants used for device model calculations.

Section 2

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
Double  *previous_voltage,          //% Pointers
        *previous_m,                //%       used
        *previous_h,                //%         for
        *previous_n,                //%           static
        *previous_voltage_temp,     //%                  variables
        *previous_m_temp,               //%           ||
        *previous_h_temp,           //%         ||
        *previous_n_temp,           //%         ||
        *lastT,                     //%         ||
        *lastCurrent;               //%         ||
double  v_rest,                     //% Membrane Resting Potential
        Cap,                        //% Membrane Capacitance
        E_Na,                       //% Sodium Potential
        E_K,                        //% Potassium Potential
        gNa,                        //% Sodium Conductance
        gK,                         //% Potassium Conductance
        v_neuron,                   //% Input voltage
        R = 8.314,                  //% Reiberg constant (joules/(mole*kelvin)).
        F = 9.648E4,                //% Faraday's constant (coulombs/mole).
        Z = 1,                      //% Sodium and potassium ionic valence.
        b = 0.02,                   //% Relative permeability of Na to K
        MO,                         //% Resting M
        HO,                         //% Resting H
        NO,                         //% Rest N
        V_r,                        //% Membrane Resting Potential
        delta_T,                    //% Timestep
        gNamax,                     //% Maximum Na conductance (S/cm^2)
        gKmax,                      //% Max K conductance (S/cm^2)
        cm,                         //% Active region capacitance (F/cm^2)
        coNa,                       //% Extracellular Na concentration (mol/L)
        ciNa,                       //% Intracellular Na concentration (mol/L)
        coK,                        //% Extracellular k concentration (mol/L)
        ciK,                        //% Intracellular K concentration (mol/L)
        VNa,                        //% Na Nernst Potential
        VK,                         //% K Nernst Potential
        ah, am, an,                 //% m, h, & n Alphas
        bh, bm, bn,                 //% m, h, & n Betas
        up, down,                   //% Used for zero  denom case error checking
        tauM, tauH, tauN,           //% Time Constants
        infM, infH, infN,           //% Final m, h, &n values
        IK,                         //% Potassium Current
        INa,                        //% Sodium Current
        I_memb,                     //% Membrane current
        m, h, n,                    //% M, H, and N values
```

Figure 7.7. Section two of device model "cfunc.mod" file.

```
Vm,                     //% Membrane Potential
gL,                     //% Leakage Conductance
Vl,                     //% Leakage Potential
Ileak,                  //% Leakage Current
Q10,                    //% Temperature scaling factor - Q10
compart_num,            //% 1-single compartment 2 for more than one
abstemp,                //% Current Absolute Temperature
V_in,                   //% Voltage In
I_in,                   //% Current In
Atotal,                 //% Membrane Total Area
V_m,                    //% Membrane Voltage
L,                      //% Neuron Cell length
aj,                     //% Neuron Cell radius
pi = 3.141592654;
```

Figure 7.7. Continued.

7.2.3 "cfunc.mod" File - Section 3

Figure 7.8 is section 3 of "cfunc.mod" file. It is the beginning of the first call to the device model by SPICE. It loads the parameters set in the ".model" line of the netlist file, if present. Any parameter not set in the ".model" line will be loaded with the default value set in the "ifspec.ifs" file or calculated, as required. Section three also calculates the area used to scale the values in the Hodgkin-Huxley model. Finally, section three calculates the resting potential if zero is passed from the "ifspec.ifs" file.

Section 3

```
/*****************************************************************/
/****                    Neuron Code Model ROUTINE                ***/

int neuron(ARGS)        /*      structure holding parms, inputs, outputs, etc.    */

{
I_in = (INPUT(a)*1e6);

if (INIT==TRUE) {       /*      First pass...allocate storage for previous value... */
```

Figure 7.8. Section three of device model "cfunc.mod" file.
/ Pull in parameters from ifspec.ifs */*

```
abstemp      = 273.15 + TEMPERATURE;
aj           = PARAM(cell_radius);
L            = PARAM(cell_length);
compart_num      = PARAM(compartment_number);

if (compart_num == 1){ /* Surface area of one cylinder, including ends  */

      Atotal = ((2*pi*aj*L*pow(100,2))+(2*pi*pow(aj,2)*pow(100,2)));
}
else{
      Atotal = (2*pi*aj*L*pow(100,2)); /*  Surface Area of cylinder for
                                           multiple compartment*/
}
cm           = (PARAM(cap)*1e6)*Atotal;
gNamax       = (PARAM(max_gna)*1e3)*Atotal;
gKmax        = (PARAM(max_gk)*1e3)*Atotal;
ciNa         = (PARAM(ci_na)*1e3);
coNa         = (PARAM(co_na)*1e3);
ciK          = (PARAM(ci_k)*1e3);
coK          = (PARAM(co_k)*1e3);
Vl           = (PARAM(v_l)*1e3);
gL           = (PARAM(g_l)*1e3)*Atotal;
V_r          = PARAM(v_rest);
Q10          = PARAM(q10);
if (V_r == 0){    /*  calculate resting potential if not defined in ifspec.ifs  */
      V_r = (((R*abstemp)/(Z*F))*log((coK + b*coNa)/(ciK +
            b*ciNa)))*1.0E3;
}
```

Figure 7.8. Continued.

7.2.4 "cfunc.mod" File - Section 4

Figure 7.9 is section 4 of the "cfunc.mod" file. It allocates static memory locations of the appropriate size for the static variables defined in the "ifspec.ifs" file. It then ties the static locations to the pointer defined in section 1.

Section 4

```
/* Allocate storage for static variables */

STATIC_VAR(previous_voltage) = (double *) malloc(sizeof(double));
        previous_voltage = STATIC_VAR(previous_voltage);

STATIC_VAR(previous_m) = (double *) malloc(sizeof(double));
        previous_m = STATIC_VAR(previous_m);

STATIC_VAR(previous_h) = (double *) malloc(sizeof(double));
        previous_h = STATIC_VAR(previous_h);

STATIC_VAR(previous_n) = (double *) malloc(sizeof(double));
        previous_n = STATIC_VAR(previous_n);

STATIC_VAR(previous_voltage_temp) = (double *) malloc(sizeof(double));
        previous_voltage_temp = STATIC_VAR(previous_voltage_temp);

STATIC_VAR(previous_m_temp) = (double *) malloc(sizeof(double));
        previous_m_temp = STATIC_VAR(previous_m_temp);

STATIC_VAR(previous_h_temp) = (double *) malloc(sizeof(double));
        previous_h_temp = STATIC_VAR(previous_h_temp);

STATIC_VAR(previous_n_temp) = (double *) malloc(sizeof(double));
        previous_n_temp = STATIC_VAR(previous_n_temp);

STATIC_VAR(lastT) = (double *) malloc(sizeof(double));
        lastT = STATIC_VAR(lastT);

STATIC_VAR(lastCurrent) = (double *) malloc(sizeof(double));
        lastCurrent = STATIC_VAR(lastCurrent);
```

Figure 7.9. Section four of device model "cfunc.mod" file.

7.2.5 "cfunc.mod" File - Section 5

Figure 7.10 is section 5 of the "cfunc.mod" file. It sets the initial conditions for the sodium and potassium Nernst potentials as well as the α and β rate constants based on the resting potential. Code is included to handle the zero denominator possibility. Section 5 also sets the final value time constants and initializes the static variables used to store previous values during the simulation.

Section 5

```
  /* Set Nernst Potentials, VNa, and VK based on ionic concentrations */

VNa = (((R*abstemp)/(Z*F))*log(coNa/ciNa))*1e3;
VK  = (((R*abstemp)/(Z*F))*log(coK/ciK))*1e3;

/* Set M, N and H initial conditions... */

 ah = 0.07*exp(-0.05*(V_r+60));
 bh = 1/(1+exp(-0.1*(V_r+30)));

/*****************************************************************/
/***            Handle indeterminate cases when denominator = 0            ***/

if (V_r == -35){
        up = V_r + 1.0E-4;
        down = V_r - 1.0E-4;
        am = (-0.1*(up+35)/(exp(-0.1*(up+35))-1) + -0.1*(down+35)/
              (exp(-0.1*(down+35))-1))/2;
}
else {
        am = -0.1*(V_r+35)/(exp(-0.1*(V_r+35))-1);
}
        bm = 4.0*exp(-(V_r+60)/18);

if (V_r == -50){
        up = V_r + 1.0E-4;
        down = V_r - 1.0E-4;
        an = (-0.01*(up+50)/(exp(-0.1*(up+50))-1) + -0.01*(down+50)/
              (exp(-0.1*(down+50))-1))/2;
}
else {
        an = -0.01*(V_r+50)/(exp(-0.1*(V_r+50))-1);
}
        bn = 0.125*exp(-0.0125*(V_r+60));


/*****************************************************************/

MO = am/(am+bm);                    /*  Final Value Time Constants  */
HO = ah/(ah+bh);
NO = an/(an+bn);
```

Figure 7.10. Section five of device model "cfunc.mod" file.

```
/* Set previous_voltage value to zero... */

*previous_voltage    = V_r;
*previous_m          = MO;
*previous_h          = HO;
*previous_n          = NO;
*lastT               = 0;
*lastCurrent         = 0;
Vm            = V_r;
}
else {                                  /*  if INIT != true...not first pass    */

        if (T(0)==0.0){
                    *previous_voltage = V_r;
                    *previous_m = MO;
                    *previous_h = HO;
                    *previous_n = NO;
                    *lastT = 0;
                    Vm = V_r;
                    *lastCurrent = 0;
        }
```

Figure 7.10. Continued.

7.2.6 "cfunc.mod" File - Section 6

Figure 7.11 is section 6 of the "cfunc.mod" file. It loads the static variable values for the model values from the last time step as well as calculates the current values of the α and β values for the current time step.

Section 6

```
        else {                          /*      if T(0) != 0.0      */
                    previous_voltage = STATIC_VAR(previous_voltage);
                    previous_m = STATIC_VAR(previous_m);
                    previous_h = STATIC_VAR(previous_h);
                    previous_n = STATIC_VAR(previous_n);
                    previous_voltage_temp =
STATIC_VAR(previous_voltage_temp);
                    previous_m_temp = STATIC_VAR(previous_m_temp);
                    previous_h_temp = STATIC_VAR(previous_h_temp);
                    previous_n_temp = STATIC_VAR(previous_n_temp);
```

Figure 7.11. Section six of device model "cfunc.mod" file.

```
                lastT      = STATIC_VAR(lastT);
                lastCurrent = STATIC_VAR(lastCurrent);
                if (T(1) != *lastT){
                        *previous_voltage = *previous_voltage_temp;
                        *previous_m = *previous_m_temp;
                        *previous_h = *previous_h_temp;
                        *previous_n = *previous_n_temp;

                        *lastT = T(1);

                        *lastCurrent = I_in/Atotal;
        }
                else{
                }
                delta_T = T(0)-T(1);

                ah = 0.07*exp(-0.05*(*previous_voltage+60));
                bh = 1/(1+exp(-0.1*(*previous_voltage+30)));

/*****************************************************************/
/***         Handle indeterminate cases when denominator = 0          ***/

                if (*previous_voltage == -35){
                                up = *previous_voltage + 1.0E-4;
                                down = *previous_voltage - 1.0E-4;
                                am = (-0.1*(up+35)/(exp(-0.1*(up+35))-1) +
                                        -0.1*(down+35)/(exp(-0.1*(down+35))-
                                        1))/2;
                }
                else {
                                am = -0.1*(*previous_voltage+35)/
                                    (exp(0.1*(*previous_voltage+35))-1);
                }
                bm = 4.0*exp(-(*previous_voltage+60)/18);

                if (*previous_voltage == -50){
                                up = *previous_voltage + 1.0E-4;
                                down = *previous_voltage - 1.0E-4;
                                an = (-0.01*(up+50)/(exp(-0.1*(up+50))-1) +
                                        -0.01*(down+50)/(exp(-0.1*(down+50))-
                                        1))/2;
                }
                else {
                                an = -0.01*(*previous_voltage+50)/
                                        (exp(-0.1*(*previous_voltage+50))-1);
                }
/*****************************************************************/
                bn = 0.125*exp(-0.0125*(*previous_voltage+60));
```

Figure 7.11. Continued.

7.2.7 "cfunc.mod" File - Section 7

Figure 7.12 is section 7 of the "cfunc.mod" file. It calculates the equations for the time constants and final m, h, and n value. It then solves the m, h, and n gating equations using the Euler method for numerical integration. This technique is valid given the very short time steps used in the SPICE netlist file. Once the gating constants are found, the sodium and potassium conductances are found, and then used to calculate the ionic currents based on Nernst potentials, conductances, cell capacitance values, and the previous membrane potential value. These currents are added to determine the total membrane current. The membrane current is then used to calculate the new membrane potential. Finally, the current values are stored into the static variables for use in the next iteration of the device model, and the membrane potential is output onto the output port "a".

Section 7

```
/*  Calculate time constants        */

                    tauM = 1/(am+bm);
                    tauH = 1/(ah+bh);
                    tauN = 1/(an+bn);

/*  Calculate final M, H, & N values    */

                    infM = am/(am+bm);
                    infH = ah/(ah+bh);
                    infN = an/(an+bn);

/*  Solving m, h, &  n  by the Euler method */

                    m = *previous_m + ((((infM - *previous_m)/tauM) * delta_T)
                        * pow(Q10,((TEMPERATURE-6.3)/10))* 1e3);
                    h  = *previous_h + ((((infH - *previous_h)/tauH) * delta_T)
                        * pow(Q10,((TEMPERATURE-6.3)/10))* 1e3);
                    n  = *previous_n + ((((infN - *previous_n)/tauN) * delta_T)
                        * pow(Q10,((TEMPERATURE-6.3)/10))* 1e3);
```

Figure 7.12. Section seven of device model "cfunc.mod" file.

```
/* Calculating conductances using m, h, and n values calculated above   */

                    gNa =(gNamax*m*m*m*h);
                    gK = (gKmax*n*n*n*n);

/* Calculating ionic currents based in conductances, Nernst potentials, previous
voltages and cell capacitance values     */

                    IK = ((VK - *previous_voltage)*gK)/cm;
                    INa = ((VNa - *previous_voltage)*gNa)/cm;

                    Ileak = ((Vl - *previous_voltage)*gL)/cm;

/* Calculating membrane current based on ionic currents   */

                    I_memb = IK + INa + Ileak + (*lastCurrent);

/* Calculating new membrane voltage based on previous voltage and current
voltage for the current timestep using the Euler method  */

                    Vm = *previous_voltage + (I_memb*delta_T*1e3);

/* Store values for next iteration  */
                    *previous_voltage_temp = Vm;
                    *previous_m_temp = m;
                    *previous_h_temp = h;
                    *previous_n_temp = n;
               }
          }

/*****************     Output to Port      ****************/

          OUTPUT(a) = Vm*1e-3;

return 0;

}
```

Figure 7.12. Continued.

7.3 Device Parameterization

There are many variables utilized within the Hodgkin-Huxley model. These variables include the Parameterization of the different variable values within the Hodgkin-Huxley model (equations 4.2 and 4.4); the internal and external concentrations of sodium and potassium and temperature values in the Nernst equations (equation 4.3); the temperature and temperature coefficient, Q_{10} value used in the rate constants to incorporate temperature effects in the model; as well as the cylindrical cell radius and length. Programming these variables into the source code of the model would not allow for making changes in the middle of the simulation run. Because the variables use default values set in the "ifspec.ifs" file embedded in the device model at compile time or user set values defined in the ".model" line of the netlist simulation file, less setup time is required to run multiple simulations using different values for different variables. Table 7.1 summarizes the parameters that can be changed between simulations as well as their

Table 7.1. List of the Hodgkin_Huxley and other associated variables, code model parameter names, default values and units

Variables	Parameter Name	Default Value	Units
Resting Potential	v_rest	* *calculated*	mV
Membrane capacitance per unit area	cap	1.0e-6	F/cm^2
Intracellular sodium ion concentration	ci_na	50.0E-3	mol/L
Extracellular sodium ion concentration	co_na	491.0E-3	mol/L
Intracellular potassium ion concentration	ci_k	400.0E-3	mol/L
Extracellular potassium ion concentration	co_k	20.11E-3	mol/L
Maximum sodium conductance per unit area	max_gna	120.0E-3	S/cm^2
Maximum potassium conductance per unit area	max_gk	36.0E-3	S/cm^2
Leakage conductance per unit area	g_l	0.3E-3	S/cm^2
Leakage potential	v_l	-49.0E-3	mV
Sodium Nernst equilibrium potentials	v_na	* *calculated*	mV
Potassium Nernst equilibrium potentials	v_k	* *calculated*	mV
Cell radius	cell_radius	10.0E-6	m
Cell length	cell_length	40.0E-6	m
Temperature coefficient	q10	3	N/A

* calculated if no value specified

default values. By configuring the parameter table section in the "ifspec.ifs" file of the code model to contain the variables of the Hodgkin-Huxley model, the values of the variables can be changed in the .model line of the circuit file. An example of the .model line used to specify parameter values is illustrated below:

```
.model neuron neuron (v_rest=-60 q10=2 cell_radius=10e-6 cell_length=40e-6 cap=1e-6)
```

The manner in which the parameters are specified is completely analogous to conventional nonlinear device model parameter specifications in SPICE. All values from table 7.1 may be specified by the user as in the above example for v_rest, q10, etc. Again, default values are used if no value is specified in the .model line.

Another functional capability of this device model, which may not specifically fall under the heading of parameterization, but fits into this section due to the flexibility it provides, involves its ability to represent multiple neurons and neuron structures by using a compartmental modeling approach. This capability is described in 7.4.

7.3.1 Temperature

How temperature affects the Hodgkin-Huxley model was discussed in 4.3.6. These affects are parameterized using the "TEMPERATURE" circuit data accessor macro explained in section 6.2.1.1.4. The temperature variables in equations 4.3 and 4.7 are tied to the SPICE simulation temperature set in the ".option" line of the netlist file used for the simulation run. Examples of the ".option" line setting the simulation temperature can be found in 7.4. The single neuron netlist file in 7.4.1 sets the simulation

temperature to 6.3°c. This value is defined in section #3 of the "cfunc.mod" source code described in section 7.1.2.3.

7.3.2 Hodgkin-Huxley Parameters

Like the temperature variable, the main Hodgkin-Huxley model variables are parameterized. The parameters defined in the device model include the Nernst equation variables used to define the sodium and potassium potentials, the leakage conductance and equilibrium potentials, the maximum sodium and potassium conductances, and the membrane capacitance. These variables have default values set in the "ifspec.ifs" file as defined in table 7.1. In the case of the Nernst potentials, the value is calculated by the device model if the user does not specify a value. The user can set the values of these parameters by using the ".model" line in the netlist simulation file. The example netlist shown in 7.5.1 sets the maximum sodium conductance (max_gna) at 80e-3 S/cm^2.

7.3.3 Other System Parameters

The remainder of the parameterized variables pertain to the modeled neuron and the initial resting membrane potential. The area of the cell modeled is defined as a cylinder. Therefore, the cell radius and length are required to calculate that area. Finally, the resting membrane potential can be set by the user in the netlist ".model" line. If it is not set in the ".model" line, it is calculated in the device model as shown in section 3 of the "cfunc.mod" source code described in section 7.1.2.3.

7.4 Compartmental Modeling

The Compartmental modeling approach to propagating action potentials may not specifically fall under the heading of parameterization. However, the approach provides great utility and flexibility for the user as a system parameter allowing the ability to represent multiple neurons and neuron structures, and is therefore, included in this chapter.

One way to describe the conduction velocity of an action potential along an axon of a given diameter is by using the Core Conductor Equation [36]. This equation relates the coupling of voltage and current along a cylindrical cell by:

$$\frac{\partial^2 V_m}{\partial z^2} = 2\pi a(r_o + r_i)J_m \qquad 7.1$$

where a is the radius of the axon; r_o is the resistance per unit length of the outer conductor; r_i is the resistance per unit length of the inner conductor; and J_m is the membrane current density. Substituting equation 4.2 into 7.1 yields:

$$\frac{1}{2\pi a(r_o + r_i)}\frac{\partial^2 V_m}{\partial z^2} = C_m\frac{\partial V_m}{\partial t} + g_K(V_m - V_k) + g_{Na}(V_m - V_{Na}) + g_L(V_m - V_L) \qquad 7.2$$

While solving this equation analytically is possible for some situations, in the case of solving voltage-dependent propagating action potential solutions, the analytical approach using the above equations must be replaced with the compartmental approach described in the literature [8;14;37;38].

The compartmental approach is based upon the ability to divide a continuous system of neurons into sufficiently small compartments, and then, make the assumption each compartment is isopotential and uniform in their size, shape, and electrical properties. This assumption makes it possible to model a nerve axon based on multiple

compartments, each based on the dynamics of the Hodgkin-Huxley model. The compartmental method reduces the non-linear partial differential equation shown in equation 7.2 to the set of ordinary differential equations where each equation represents one compartment of the axon of length "Δx" [39]. Each compartment is connected in series by resistors acting as the axoplasm resistance (R_i). Figure 7.13 shows the "j_{th}" element of an axon with the previous and next isopotential cell, delineated by the dotted lines, connected through the axonal resistances. Section 7.5.2 describes a 10-neuron axon using the compartmental model described above.

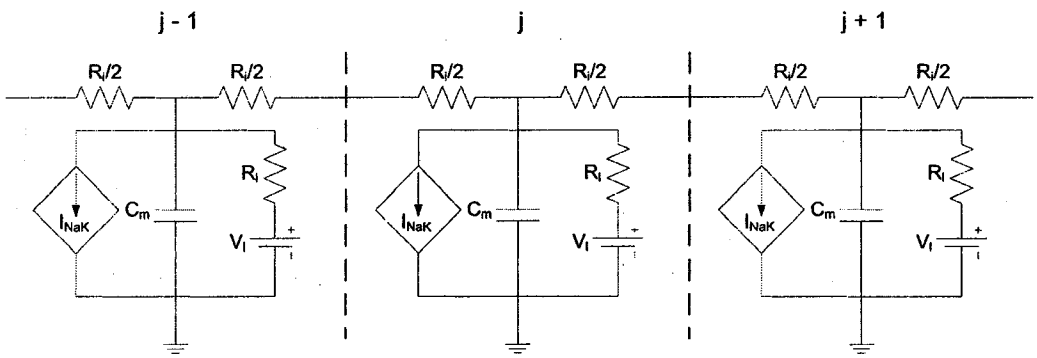


Figure 7.13. The Hodgkin-Huxley equivalent circuit diagram depicting a patch of a neuronal cell membrane in terms of ionic conductances (G), current densities (I), ionic potentials (E) and membrane capacitance s (C_m).

7.5 Example Netlist Files

In order to show the utility of the parameterized device model within a SPICE simulation, two example netlists are included for reference in the parameterization discussion. The first netlist is a simulation of a single neuron, while the second example

netlist utilizes the compartmental approach discussed in 7.3 to connect 10 neurons together to form one axon.

7.5.1 Single Neuron

The netlist showed in figure 7.14 uses the neuron device model to simulate an injected current of 1nA into a single neuron. The netlist utilizes the parameterization of the device model by setting the resting membrane potential, temperature coefficient, cell radius, cell length, and maximum sodium conductance parameters in the ".model" line as well as the temperature parameters in the ".options" line. All other model parameters are either set in the "ifspec.ifs" file or calculated within the model itself. The output shown in figure 7.15 is a screen capture of the SPICE OPUS plot created by the SPICE simulation using the "plot v(1)" command for the output of the netlist showed in figure 7.14.

```
Neuron Test File
I 0 1 pulse(0 1e-9 5e-3 0 0 5e-3 20e-3)
a1 1 neuron
.model neuron neuron (v_rest=-61 q10=3 cell_radius=10e-6
          cell_length=80E-06 max_gna=115e-3)
.options temp=6.3 tnom=6.3
.tran 1e-6 20e-3
.END
```

Figure 7.14. Example netlist for a 1nA injected current into a single neuron.

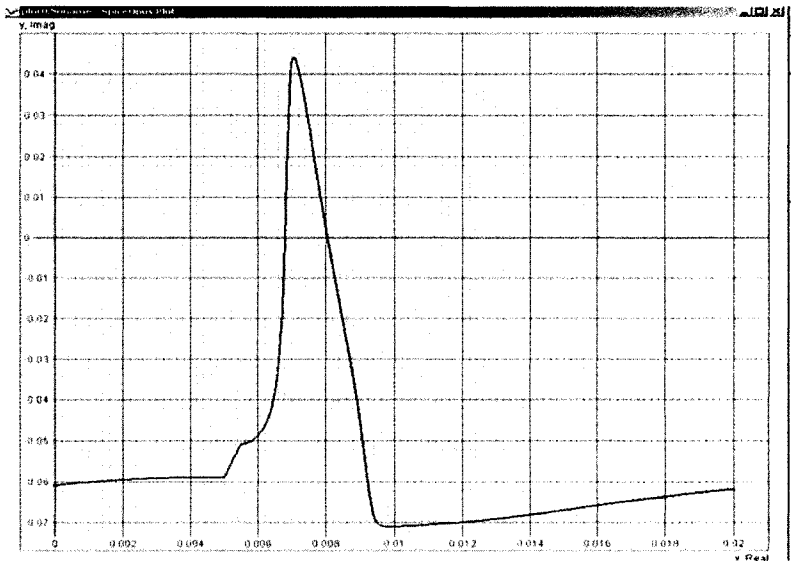


Figure 7.15. Output of the single neuron action potential simulation utilizing the code from figure 7.14.

7.5.2 Ten Compartment Axon

The netlist showed in figure 7.16 uses the neuron device model to simulate an injected current of 50nA into an axon made up of 10 neurons using the compartmental modeling approach. Each device model is separated by a resistor acting as the intracellular resistance. The netlist utilizes the parameterization of the device model by setting the resting membrane potential, temperature coefficient, cell radius, cell length, and compartment number in the ".model" line as well as the temperature parameters in the ".options" line. All other model parameters are either set in the "ifspec.ifs" file or calculated within the model itself. The output shown in figure 7.17 is a screen capture of the SPICE OPUS plot created by the SPICE simulation using the "plot v(1) v(2) v(3) v(4) v(5) v(6) v(7) v(8) v(9) v(10)" command for the output of the netlist showed in figure 7.16.

```
Neuron Test File
I 0 1 pulse(0 50e-9 5e-3 0 0 1e-3 25e-3)
a1 1 neuron
r1 1 2 11.3E+06
a2 2 neuron
r2 2 3 11.3E+06
a3 3 neuron
r3 3 4 11.3E+06
a4 4 neuron
r4 4 5 11.3E+06
a5 5 neuron
r5 5 6 11.3E+06
a6 6 neuron
r6 6 7 11.3E+06
a7 7 neuron
r7 7 8 11.3E+06
a8 8 neuron
r8 8 9 11.3E+06
a9 9 neuron
r9 9 10 11.3E+06
a10 10 neuron
r10 10 11 11.3E+06
.model neuron neuron (v_rest=-60 q10=3 cell_radius=5e-6
                      cell_length=2.5E-03 compartment_number=2)
.options temp=6.3 tnom=6.3
.tran 1e-6 25e-3
.END
```

Figure 7.16. Example netlist for a 50 nA injected current into an axon made up of 10 Hodgkin-Huxley modeled neurons.

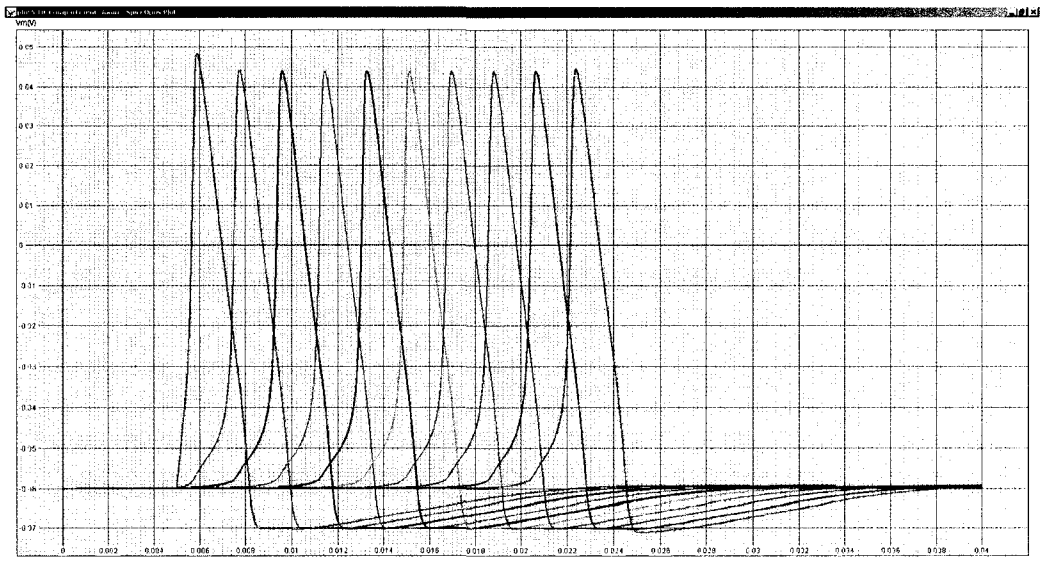


Figure 7.17. Output of the 10-compartment axon simulation showing propagation of the initial action potential utilizing the code from figure. 7.16

CHAPTER 8

DEVICE MODEL VALIDATION

In order to validate the device model associated with this dissertation, multiple simulations were compared to existing results in the literature. Major aspects of the Hodgkin-Huxley model were first chosen as points of interest for validation. These validation points are detailed in 4.3.1-4.3.5. Once the basic validation of the Hodgkin-Huxley was complete, the validation was shifted to temperature as detailed in 4.3.6. Finally, two types of temperature blocks are compared to the device model results for the same phenomena. The validation was completed using both SPICE OPUS and NGSPICE implementations of the device model which demonstrates the portability of the device model within SPICE simulation software containing the XSPICE Code Model toolkit.

8.1 Comparison to Standard Hodgkin-Huxley Values

The first area of validation for the device model created for this dissertation involves the major aspects of the Hodgkin-Huxley model discussed in sections 4.3.1-4.3.6 to include m, h, and n values, repetitive activity, accommodation, anode-break

excitation, subthreshold oscillations, and the affects of temperature on action potential generation.

8.1.1 m, h, & n Value Validation

The first model aspect used for validation involves the actual values of the gating variables, m, h, and n at rest as well as throughout the generation of an action potential. As discussed in section 3, these gating variables are tied to the m, h, and n gates which react to depolarization of the cellular membrane at the initiation of an action potential. The "m" sodium activation gate reacts quickly, opening once a depolarization of 10-20 mV occurs. The "h" activation gate begins to close upon depolarization; however, it closes more slowly compared to the "m" gate activity. The "h" gate remains closed until the refractory period has passed Finally, the potassium "n" gate begins to open once sufficient depolarization has occurred to start the action potential, but like the sodium "h" gate, it is also slow to react. The n gate remains open past the point of repolarization to the resting potential of the cellular membrane. This phenomenon is illustrated on page 202, figure 4.32 in Weiss [18]. Figure 8.1 shows the device models results for the m, h, and n gates throughout the action potential generation process.

8.1.2 Repetitive Activity Validation

As discussed in section 4.3.2, studies have shown that a suprathreshold current applied to a nerve cell over time will cause multiple action potentials [22]. When currents slightly above the cell's threshold are used, only a single action potential is created. This is simulated in panel A, figure 8.3. When higher continuous current amplitudes are applied, multiple action potentials are seen with the latter action

potentials' maximum amplitude being less than the initial action potential. This is simulated in panels B-E in figure 8.3. Ultimately, action potentials caused by higher currents are blocked demonstrating a property called a depolarization block. This is simulated in panel F of figure 8.3. This effect is caused by the continuous current's ability to decrease h and increase n over time. These changes affect the sodium and potassium conductances which in turn reduces the action potential amplitude ultimately

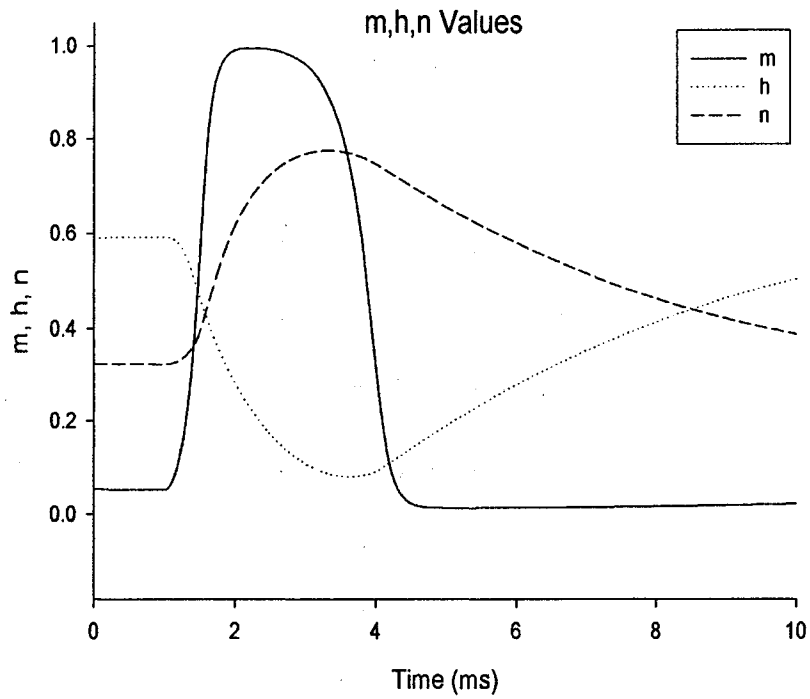


Figure 8.1. Device model results for m, h, and n computed using a 0.5 msec current stimulus of 10 nA and the standard Hodgkin-Huxley parameters from table 4.1.

resulting in the block. This affect is called repetitive activity, and the Hodgkin-Huxley model exhibits the basics of this phenomenon. An illustration for different stimulus amplitudes can be found on page 232, figure 4.60 in Weiss (included in appendix C) [18]. Figure 8.2 shows the SPICE netlist used to simulate the device model results seen in figure 8.3 using the identical suprathreshold currents used in Weiss.

8.1.3 Refractory Period Validation

As discussed in section 3.6, the sixth property of action potentials is associated with the cells refractory period. Once the slow acting "h" sodium gates close, a subsequent action potential in the same cell is prohibited due to the inability of sodium ions to enter the cell and begin the depolarization phase again. The time it takes the sodium "h" gates to reopen and therefore allow a new action potential to begin is known as the refractory period after the current action potential has ended [21]. Two related examples are provided for the validation of this portion of the device model. Figure 8.4 shows the SPICE netlist used for the related simulation. Figure 8.6 shows the actual SPICE OPUS graphs of the refractoriness noted in the original Hodgkin-Huxley work [9].

```
Repetitive Activity Neuron Test File
I 0 1 pulse(0 5e-6 0 0 0 40e-3 50e-3)
a1 1 neuron
.model neuron neuron (v_rest=-61 q10=3 compartment_number=0)
.options temp=6.3 tnom=6.3
.tran 1e-6 50e-3
.END
```

Figure 8.2. Hodgkin-Huxley repetitive activity comparison netlist.

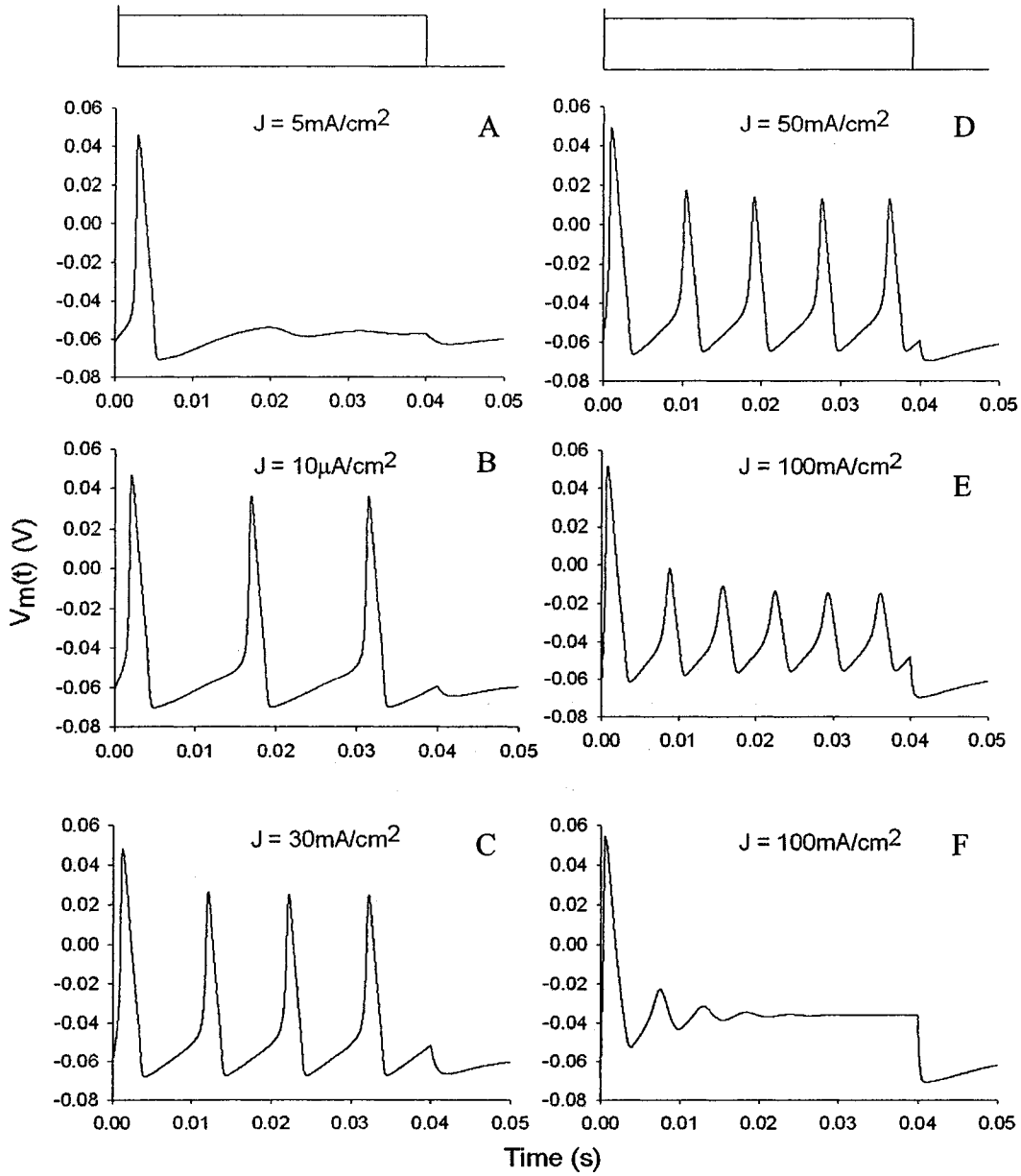


Figure 8.3. Actual device model results showing action potential repetitive activity using different current injections over a 40 ms time period—comparison with Weiss, page 232, figure 4.60 (graph provided in appendix C).

A 15 nA/cm^2 current is injected into a neuron at t=0 in all three graphs. The left-most graph in figure 8.6 shows a second current injection of 90 nA/ cm^2 at 4 ms after the first current injection with no action potential generation. The center graph in figure 8.6

shows a second current injection of 90 nA/ cm^2 at 5 ms after the first current injection with a resultant action potential of reduced amplitude. The right-most graph in figure 8.6 shows a second current injection of 90 nA/ cm^2 at 11 ms after the first current injection with a full action potential generated. Figure 8.5 shows the netlist file used to simulate the related, basic simulation seen in figure 8.7. The simulation contains three current injections of equal magnitude injected at 10 ms intervals. The first and third action potential fires, but due to refractoriness, the second injection does not fire.

```
Hodgkin-Huxley Refractoriness Neuron Test File
I1 0 1 pulse(0 15e-9 0e-3 0 0 1e-3 25e-3)
*I2 0 1 pulse(0 90e-9 4e-3 0 0 1e-3 25e-3)
*I2 0 1 pulse(0 90e-9 5e-3 0 0 1e-3 25e-3)
*I2 0 1 pulse(0 90e-9 11e-3 0 0 1e-3 25e-3)
a1 1 neuron2
.model neuron2 neuron2 (v_rest=-60 q10=3 cell_radius=5e-6
cell_length=2.5E-03)
.options temp=9 tnom=9
.tran 1e-6 25e-3
.END
```

Figure 8.4. Hodgkin-Huxley refractory period comparison netlist—for each time value simulated, the appropriate current input line should be uncommented.

```
Basic Refractory Period Neuron Test File

I1 0 1 pulse(0 15e-9 10e-3 0 0 1e-3 10e-3)
a1 1 neuron
.model neuron neuron (v_rest=-60 q10=3 cell_radius=5e-6
                      cell_length=2.5E-03)
.options temp=6.3 tnom=6.3
.tran 1e-6 40e-3

.END
```

Figure 8.5. Hodgkin-Huxley repetitive activity comparison netlist.

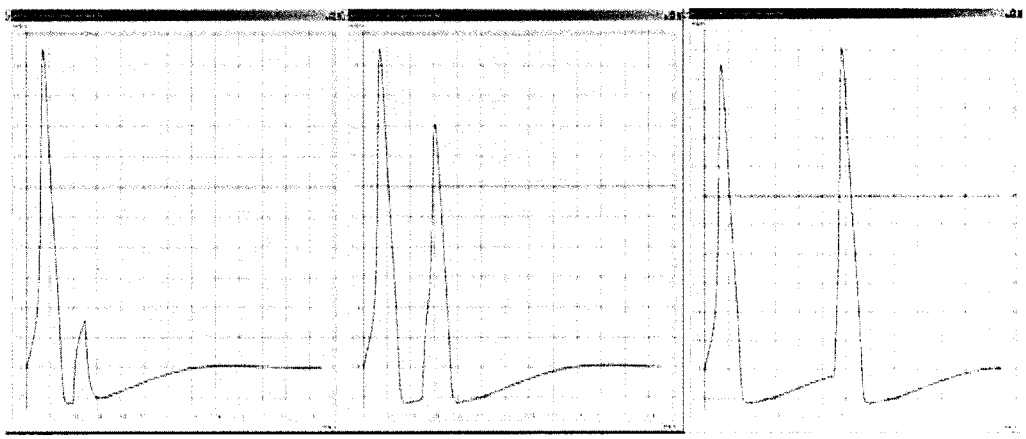


Figure 8.6. A 15 nA/cm^2 current is injected into a neuron at t=0 in all three graphs. The left-most graph shows a second current injection of 90 nA/ cm^2 at 4 ms after the first current injection with no action potential generation. The center graph shows a second current injection of 90 nA/ cm^2 at 5 ms after the first current injection with an action potential of reduced amplitude is generated. The right-most graph shows a second current injection of 90 nA/ cm^2 at 11 ms after the first current injection with a full action potential generated.

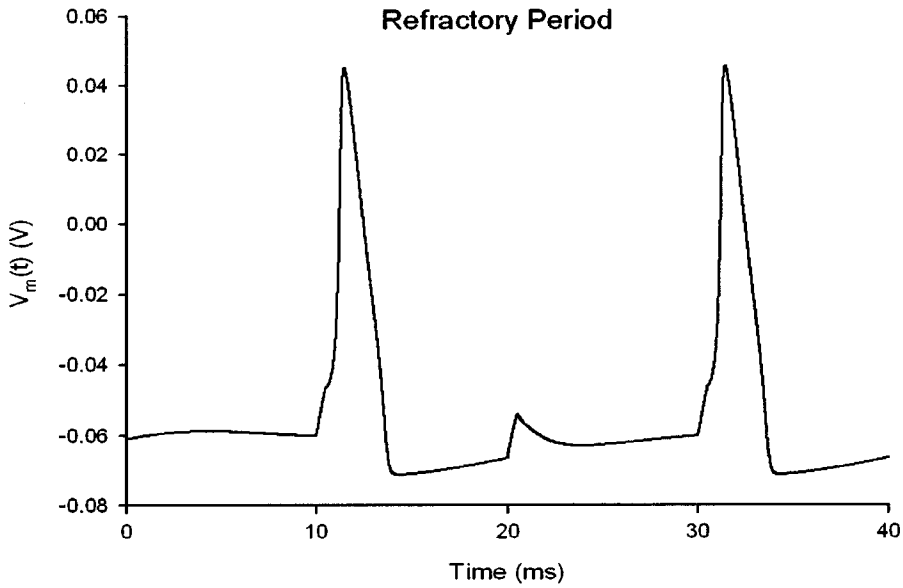


Figure 8.7. Device model results for the Basic refractory period simulation completed using the netlist in figure 8.5.

8.1.4 Anode-Break Excitation Validation

As discussed in section 4.3.4, another phenomenon present in nerve cells occurs when a hyperpolarizing current is passed into a nerve cell for a period of time long enough for the membrane potential and all gating variables to stabilize at their steady-state values. When the current is subsequently removed from the cell, an action potential is generated. This is known as an anode-break excitation, and it occurs due to the increase in h and the decrease in n which affects the potassium conductance. This lowers the threshold necessary for initiating an action potential. This new threshold is achieved when the hyperpolarized current is removed, and the cell is repolarized to its normal resting potential [18]. The calculated anode-break excitation phenomenon is shown in Weiss on page 230, figure 4.59 (included in appendix C) [18]. Figure 8.8 shows the netlist file used to simulate device model's results shown in figure 8.9 using the same values given in Weiss. Temperature was set to 18.5°C and the maximum sodium conductance was set to 160 mS/cm^2.

```
Anode-Break Neuron Test File

I 0 1 pulse(0 -9e-9 0e-3 0 0 15e-3 30e-3)
a1 1 neuron
.model neuron neuron (v_rest=-60 q10=3 cell_radius=5e-6
                      cell_length=2.5E-03 max_gna=160e-3)
.options temp=18.5 tnom=18.5
.tran 1e-6 30e-3
.END
```

Figure 8.8. Hodgkin-Huxley anode-break comparison netlist.

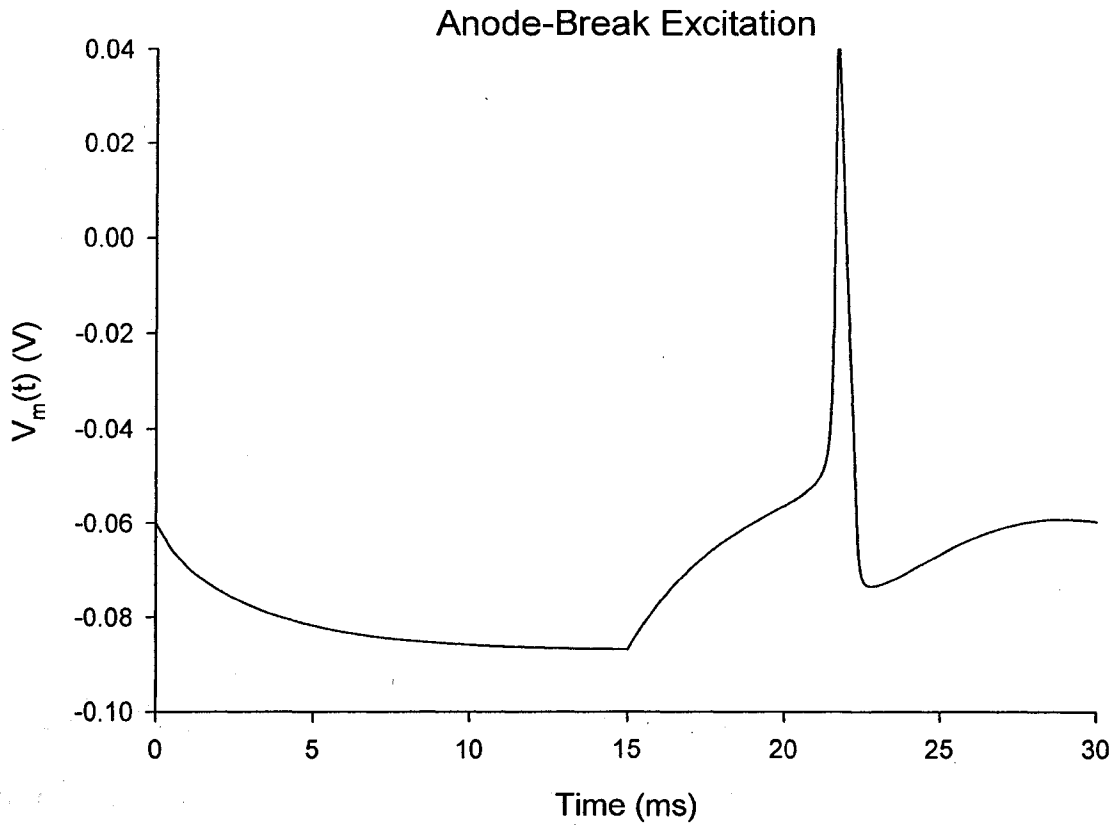


Figure 8.9. Anode-break device model results—comparison with Weiss, page 230, figure 4.59 (graph provided in appendix C).

8.1.5 Accommodation Validation

As discussed in section 4.3.3, if a current stimulus is applied to a nerve cell below the threshold value for that cell, and then slowly increased over time, no action potential is generated even though the threshold value for the cell is subsequently exceeded. Known as accommodation, this phenomenon has been studied since the mid nineteenth century [23]. Later studies using the space clamped squid axon apply ramped currents of differing slopes [22;24]. These studies found slope values below which no action potential was generated despite the threshold value of current being exceeded under

normal step conditions. The Hodgkin-Huxley model does not exhibit accommodation at the standard parameters of the model; however, if the maximum sodium conductance ($\bar{g}_{Na}$) is lowered to 80.0E-3 S/cm^2, the Hodgkin-Huxley model does exhibit the accommodation phenomenon [18]. The accommodation effect is shown on page 228, figure 4.56 in Weiss (included in appendix C) [18]. Figure 8.10 shows the netlist file used to simulate the device model's results shown in figure 8.11 using the different $\bar{g}_{Na}$ value of 80.0E-3 S/cm^2.

```
Accommodation Neuron Test File
I 0 1 pulse(0 800e-6 0 2000e-3 0 2000e-3 2000e-3)
a1 1 neuron
.model neuron neuron (v_rest=-61 q10=3 compartment_number=0
                      max_gna=80e-3)
.options temp=18.5 tnom=18.5
.tran 1e-6 2000e-3
.END
```

Figure 8.10. Hodgkin-Huxley repetitive activity comparison netlist.

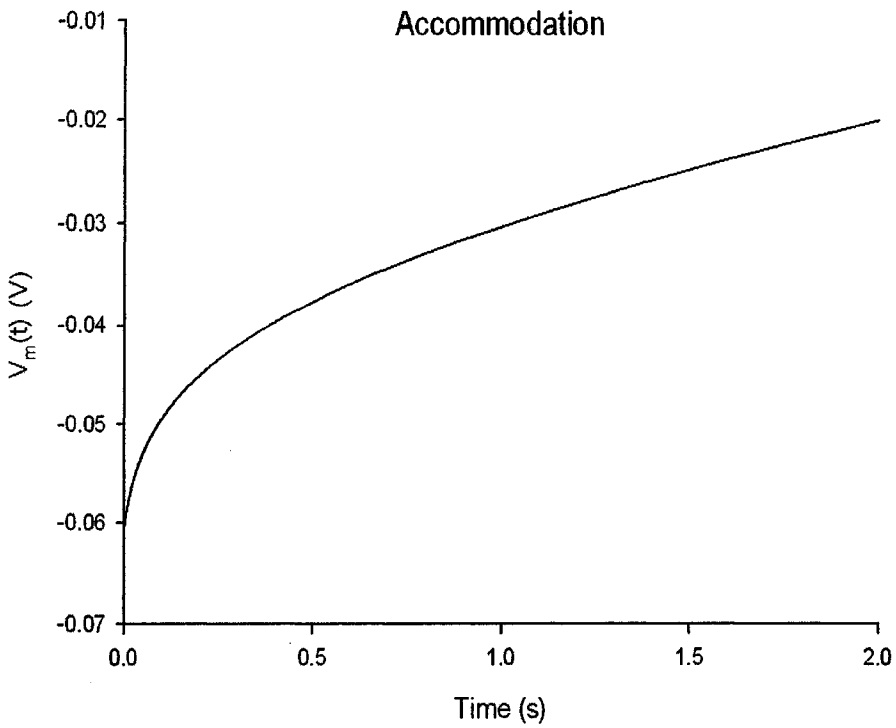


Figure 8.11. Accommodation device model results—comparison with Weiss, page 228, figure 4.56 (graph provided in appendix C).

8.1.6 Subthreshold Oscillations Validation

When a continuous pulse of a relatively small subthreshold current is passed into a nerve cell, the membrane potential exhibits highly damped oscillations [9;25]. This effect happens for both positive and negative currents. Subthreshold oscillations are seen in the Hodgkin-Huxley model. The calculated subthreshold oscillations phenomenon is shown on page 233, figure 4.61 of Weiss [18]. Figure 8.12 shows the netlist file used to simulate the device model's subthreshold oscillation results shown in figure 8.13.

```
Subthreshold Oscillations Neuron Test File
I 0 1 pulse(0 -1.49e-6 1e-3 0 0 15e-3 25e-3)
a1 1 neuron2
.model neuron2 neuron2 (v_rest=-60.8 q10=3 compartment_number=0)
.options temp=18.5 tnom=18.5
.tran 1e-6 25e-3
.END
```

Figure 8.12. Subthreshold oscillations comparison netlist.

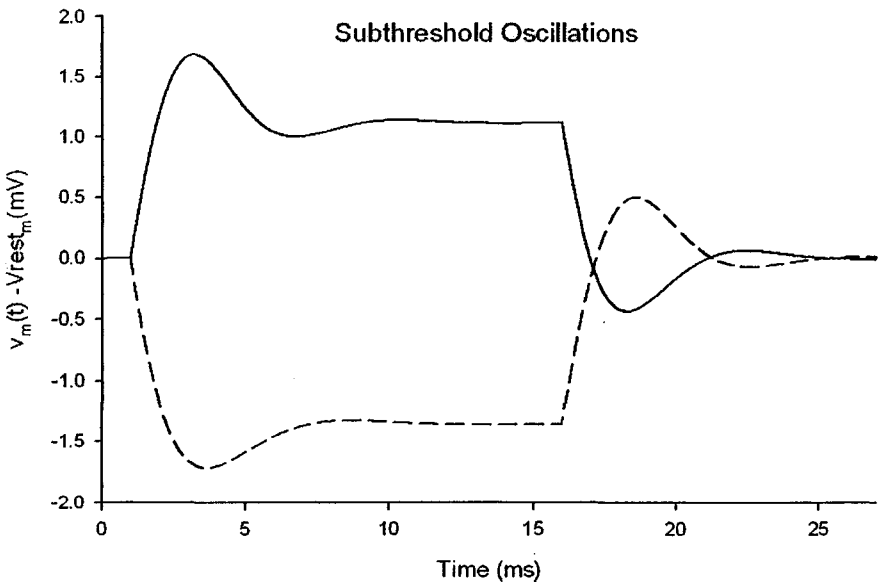


Figure 8.13. Subthreshold oscillations device model results—comparison with Weiss, page 233, figure 4.61.

8.1.7 Temperature Effects Validation

As discussed in section 4.3.6, temperature dependence in the Hodgkin-Huxley model is included in two ways [9]. The temperature dependence is included within the Nernst equilibrium potential equations seen in equation 2.1. The Nernst potential of the ions involved is proportional to absolute temperature. Temperature also has an impact on the

Hodgkin-Huxley model through the rate constants. The m, h, and n differential equations are scaled by a constant temperature factor, K_t [18]. Equation 4.6 includes the temperature factor in the m, h, and n equations. The affect of temperature on the Hodgkin-Huxley modeled action potential is shown on page 242, figure 4.68 in Weiss [18]. Figure 8.14 shows the device model's results using the same temperatures used in Weiss. As the simulation temperature is increased, the action potential rate is increased. This increase continues from 0°C up until 22°C. At this point the temperature affects the action potential to the point of blocking the action potential from firing at all.

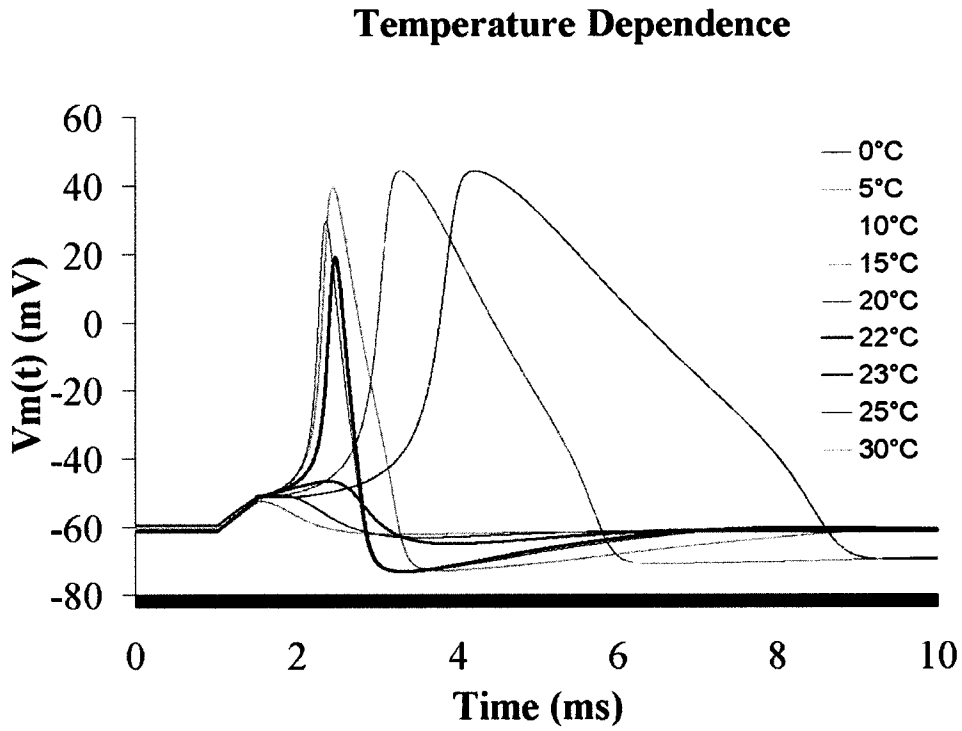


Figure 8.14. Device model results showing the effect of temperature on the Hodgkin-Huxley model—comparison with Weiss, page 242, figure 4.68.

8.2 Comparison to Different Blocking Phenomena

The last two phenomenon used to validate the device model for this dissertation include two action potential blocking phenomenon. The first is the temperature block demonstrated in Weiss [18]. The second block used to validate the device model involves a direct current block reported by Bhadra [19].

8.2.1 Temperature Block Between 22°C and 23°C Validation

To compare the thermal block reported in Weiss, a pair of simulations were implemented with a stimulus current density of 20 $\mu m/cm^2$ and the default Hodgkin-Huxley parameter values. As discussed in Weiss, thermal block occurs in the Hodgkin-Huxley model between 22-23°C [18]. Using our model, Figure 8.15 shows the action potential fires at 22°C, but no action potential occurs at 23°C.

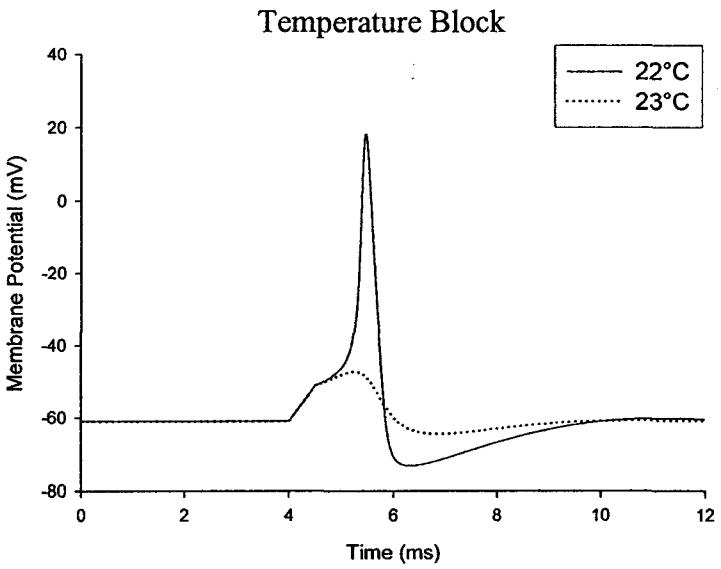


Figure 8.15. Device model results showing the thermal block between 22°C and 23°C.

8.2.2 DC Conduction Block Comparison

There are numerous studies presented in the literature that demonstrate the ability of an injected electrical current to block the conduction of action potentials. These studies have used different waveforms (i.e. direct current, high frequency biphasic and sinusoidal current sources) to perform the conduction block [15;19;40;41]. A DC conduction block was chosen to validate the neuron device model [19]. In the study, a conduction block was implemented using a 50 mm axon with a diameter of 10 μm divided into 21 compartments. A blocking current of 250 nA was initiated in compartment #10 at time = 10msec and maintained for 80 msec. At time = 40msec, a 50 nA test pulse was initiated at compartment #1. Compartment #21 was monitored for propagation of the action potential from the test pulse. The block completely stopped the test pulse from passing compartment #10. Figure 8.16 illustrates the simulation setup. The netlist used to compare the direct current study is shown in Figure 8.18. Figure 8.17 shows the results for the simulation run using the device model. The blocking current initiated at compartment #10 at 10 ms and continued for 80 ms. The propagated pulse from the block initiation, referred to as the "make" pulse is recorded at the test pulse and monitor sites. The stimulus pulse is fired at compartment #1 at 40 ms. No pulse is generated at the monitor site at compartment #21 from the test pulse due to the direct current block initiated at the blocking site.

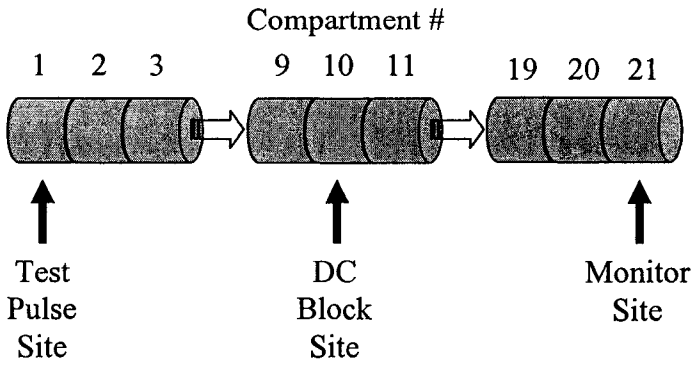


Figure 8.16. Illustration of the 21 compartment axon and sites used during the dc conduction block.

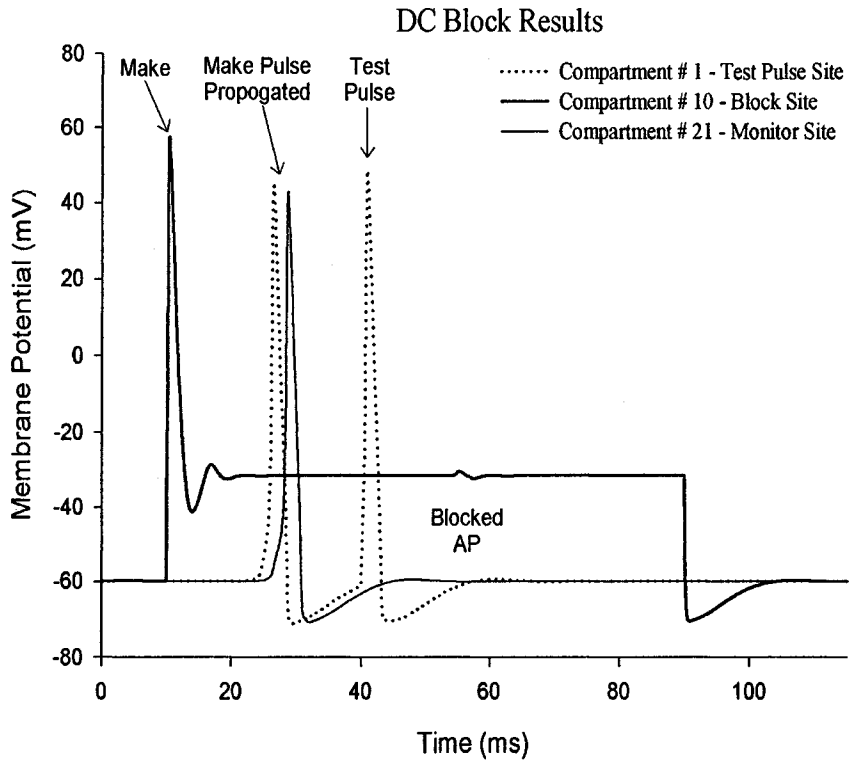


Figure 8.17. A conduction block implemented in a 50 mm axon with a diameter of 10 μm divided into 21 compartments. A blocking current of 250 nA is initiated at compartment #10 at 10 ms and continued for 80 ms causing a propagated pulse, known as a "make" pulse, seen at the test and monitor site. A 50 nA test pulse is fired at compartment #1 at 40 ms. No pulse is generated at the monitor site at compartment #21 from the test pulse due to the direct current block initiated at the blocking site.

```
Neuron Test File
************************************************
*    Test Pulse
ITest 0 1 pulse(0 50e-9 40e-3 0 0 1e-3 100e-3)
************************************************
*    Block Input  pulse
IBlock 0 10 pulse(0 250e-9 10e-3 0 0 80e-3 100e-3)
************************************************
a1 1 neuron
r1 1 2 11.3E+06
a2 2 neuron
r2 2 3 11.3E+06
a3 3 neuron
r3 3 4 11.3E+06
a4 4 neuron
r4 4 5 11.3E+06
a5 5 neuron
r5 5 6 11.3E+06
a6 6 neuron
r6 6 7 11.3E+06
a7 7 neuron
r7 7 8 11.3E+06
a8 8 neuron
r8 8 9 11.3E+06
a9 9 neuron
r9 9 10 11.3E+06
a10 10 neuron
r10 10 11 11.3E+06
a11 11 neuron
r11 11 12 11.3E+06
a12 12 neuron
r12 12 13 11.3E+06
a13 13 neuron
r13 13 14 11.3E+06
a14 14 neuron
r14 14 15 11.3E+06
a15 15 neuron
r15 15 16 11.3E+06
a16 16 neuron
r16 16 17 11.3E+06
a17 17 neuron
r17 17 18 11.3E+06
a18 18 neuron
r18 18 19 11.3E+06
a19 19 neuron
r19 19 20 11.3E+06
a20 20 neuron
r20 20 21 11.3E+06
a21 21 neuron
r21 21 22 11.3E+06
.model neuron2 neuron (v_rest=-60 q10=3 cell_radius=5e-6
                    cell_length=2.5E-03 compartment_number=2)
.options temp=6.3 tnom=6.3
.tran 1e-6 100e-3
.END
```

Figure 8.18. Netlist used to simulate the 21 compartment dc conduction block.

CHAPTER 9

CONCLUSION

We have presented a novel and improved active membrane device model in SPICE. While some implementations made use of equivalent circuit models to implement the Hodgkin-Huxley system of equations, this model is similar to a technique developed earlier which altered the source code of an early version of the University of Berkley's SPICE2G to implement the differential equations in the Hodgkin-Huxley model [7]. The improvement associated with our approach involves implementing the device model using a standard feature of most versions of SPICE. This device model is implemented using the Code-model Toolkit first created in the XSPICE version of SPICE. This toolkit allows for adding new models to existing versions of SPICE without changing the source code. The above approach incorporates within the SPICE code model the flexibility of changing parameters within the SPICE ".cir" files at runtime instead of hard coding the parameters into the source code. Once created, the SPICE cfunc.mod and ifspec.ifs files can be easily installed on any computer running a version of SPICE which implements the XSPICE functionality (e.g. NGSPICE, TCLSPICE, and SPICE OPUS).

The functionality of this device model was exhibited by demonstrating the ability of the model to reproduce documented results for a select group of characteristics found

in the Hodgkin-Huxley model. Also, two blocking phenomenon were simulated using the device model and compared with simulations and data from the literature. The first block was the natural thermal block found between 22°C and 23°C as shown in Weiss [18]. The last comparison used to validate the device model was the direct current block documented by Bhadra [19]. Both simulations were implemented using the same device model. Parameter values were altered by way of a simple alteration to the SPICE netlist file.

The novelty and robustness of the modeling approach described herein is based on the ease of implementation. A wide variety of active membranes can be simulated using this code model approach. These biologically realistic components can be integrated with artificial electronic components allowing for the simulation of hybrid neural-electronic circuitry under the SPICE simulation platform. These types of hybrid circuit simulations are not currently achievable using other neural simulators such as NEURON or GENESIS.

The logical next step for this research would be to use the device model to simulate possible integration schemes between nerve cells and electronic components; however, this work is more of an example of what the code model toolkit functionality can do than an end-all-be-all for neuronal modeling. Further research implementing other biological processes which may be integrated with artificial electronic components is wide open given the flexibility of the code model functionality.

APPENDIX A

IFSPEC.IFS SOURCE CODE

```
/*================================================

Program Name - "ifspec.ifs"

AUTHOR
Anthony S. Carver, Louisiana Tech, College of Engineering and Sciences

MODIFICATIONS
None

SUMMARY
   This file contains the interface specification file for the
   Neuron code model.

================================================*/


NAME_TABLE:

C_Function_Name:        neuron
Spice_Model_Name:       neuron
Description:            "Hodgkin-Huxley Code Model"

PORT_TABLE:

Port_Name:              a
Description:            "Neuron Input/Output port"
Direction:              inout
Default_Type:           h               /*Current Controlled Voltage Source*/
Allowed_Types:          [h]
Vector:                 no
Vector_Bounds:          -
Null_Allowed:           no

PARAMETER_TABLE:
Parameter_Name:         v_rest                  cap
Description:            "Resting voltage"       "Capacitance Value"
Data_Type:              real                    real
Default_Value:          0                       1.0E-6
Limits:                 -                       -
Vector:                 no                      no
Vector_Bounds:          -                       -
Null_Allowed:           yes                     yes

PARAMETER_TABLE:
Parameter_Name:         ci_na                   co_na
Description:            "Na intracell conc"     "Na intracell conc"
Data_Type:              real                    real
Default_Value:          50.0E-3                 491.0E-3
Limits:                 -                       -
Vector:                 no                      no
Vector_Bounds:          -                       -
Null_Allowed:           yes                     yes
```

PARAMETER_TABLE:

Parameter_Name:	ci_k	co_k
Description:	"K intracell conc"	"K extracell conc"
Data_Type:	real	real
Default_Value:	400.0E-3	20.11E-3
Limits:	-	-
Vector:	no	no
Vector_Bounds:	-	-
Null_Allowed:	yes	yes

PARAMETER_TABLE:

Parameter_Name:	max_gna	max_gk
Description:	"Max Sodium Conductance"	"Max Potassium Conductance"
Data_Type:	real	real
Default_Value:	120.0E-3	36.0E-3
Limits:	-	-
Vector:	no	no
Vector_Bounds:	-	-
Null_Allowed:	yes	yes

PARAMETER_TABLE:

Parameter_Name:	g_l	v_l
Description:	"Leakage conductance"	"Leakage voltage"
Data_Type:	real	real
Default_Value:	0.3E-3	-49.0E-3
Limits:	-	-
Vector:	no	no
Vector_Bounds:	-	-
Null_Allowed:	yes	yes

PARAMETER_TABLE:

Parameter_Name:	cell_radius	cell_length
Description:	"neuron radius-meters"	"neuron length-meters"
Data_Type:	real	real
Default_Value:	1	1
Limits:	-	-
Vector:	no	no
Vector_Bounds:	-	-
Null_Allowed:	yes	yes

PARAMETER_TABLE:

Parameter_Name:	q10	compartment_number
Description:	"temperature factor"	"1 = single compartment, 2 = more than 1"
Data_Type:	real	int
Default_Value:	3	1
Limits:	-	-
Vector:	no	no
Vector_Bounds:	-	-

Null_Allowed:	yes	yes

STATIC_VAR_TABLE:
Static_Var_Name: previous_voltage
Data_Type: pointer
Description: "iteration holding previous voltage value"

STATIC_VAR_TABLE:
Static_Var_Name: previous_m
Data_Type: pointer
Description: "iteration holding previous m value"

STATIC_VAR_TABLE:
Static_Var_Name: previous_h
Data_Type: pointer
Description: "iteration holding previous h value"

STATIC_VAR_TABLE:
Static_Var_Name: previous_n
Data_Type: pointer
Description: "iteration holding previous n value"

STATIC_VAR_TABLE:
Static_Var_Name: previous_current
Data_Type: pointer
Description: "iteration holding previous current value"

STATIC_VAR_TABLE:
Static_Var_Name: previous_voltage_temp
Data_Type: pointer
Description: "iteration holding variable for limiting"

STATIC_VAR_TABLE:
Static_Var_Name: previous_m_temp
Data_Type: pointer
Description: "iteration holding previous m value"

STATIC_VAR_TABLE:
Static_Var_Name: previous_h_temp
Data_Type: pointer
Description: "iteration holding previous h value"

STATIC_VAR_TABLE:
Static_Var_Name: previous_n_temp
Data_Type: pointer
Description: "iteration holding previous n value"

STATIC_VAR_TABLE:
Static_Var_Name: previous_current_temp
Data_Type: pointer
Description: "iteration holding previous current value"

STATIC_VAR_TABLE:
Static_Var_Name: lastT
Data_Type: pointer
Description: "iteration holding previous T(0)"

STATIC_VAR_TABLE:
Static_Var_Name: lastCurrent
Data_Type: pointer
Description: "Holds input current between iterations"

APPENDIX B

CFUNC.MOD SOURCE CODE

```
/*==============================================

CFUNC.mod

AUTHOR
Anthony S. Carver, Louisiana Tech, College of Engineering and Sciences

MODIFICATIONS
None

SUMMARY
   This file contains the model-specific routines used to
   functionally describe the Neuron code model.

==============================================*/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
Double  *previous_voltage,          //% Pointers
        *previous_m,                //%       used
        *previous_h,                //%         for
        *previous_n,                //%           static
        *previous_voltage_temp,     //%                 variables
        *previous_m_temp,               //%           ||
        *previous_h_temp,           //%         ||
        *previous_n_temp,           //%         ||
        *lastT,                     //%         ||
        *lastCurrent;               //%         ||
double  v_rest,                     //% Membrane Resting Potential
        Cap,                        //% Membrane Capacitance
        E_Na,                       //% Sodium Potential
        E_K,                        //% Potassium Potential
        gNa,                        //% Sodium Conductance
        gK,                         //% Potassium Conductance
        v_neuron,                   //% Input voltage
        R = 8.314,                  //% Reiberg gas constant
                                    //%         (joules/(mole*kelvin)).
        F = 9.648E4,                //% Faraday's constant (coulombs/mole).
        Z = 1,                      //% Sodium and potassium ionic valence.
        b = 0.02,                   //% Relative permeability of Na to K
        MO,                         //% Resting M
        HO,                         //% Resting H
        NO,                         //% Rest N
        V_r,                        //% Membrane Resting Potential
        delta_T,                    //% Timestep
        gNamax,                     //% Maximum Na conductance (S/cm^2)
        gKmax,                      //% Max K conductance (S/cm^2)
        cm,                         //% Active region capacitance (F/cm^2)
        coNa,                       //% Extracellular Na concentration (mol/L)
        ciNa,                       //% Intracellular Na concentration (mol/L)
        coK,                        //% Extracellular k concentration (mol/L)
        ciK,                        //% Intracellular K concentration (mol/L)
```

```
        VNa,                    //% Na Nernst Potential
        VK,                     //% K Nernst Potential
        ah, am, an,             //% m, h, & n Alphas
        bh, bm, bn,             //% m, h, & n Betas
        up, down,               //% Used for zero  denom case error checking
        tauM, tauH, tauN,       //% Time Constants
        infM, infH, infN,       //% Final m, h, &n values
        IK,                     //% Potassium Current
        INa,                    //% Sodium Current
        I_memb,                 //% Membrane current
        m, h, n,                //% M, H, and N values
        Vm,                     //% Membrane Potential
        gL,                     //% Leakage Conductance
        Vl,                     //% Leakage Potential
        Ileak,                  //% Leakage Current
        Q10,                    //% Temperature scaling factor - Q10
        compart_num,            //% Equals 1-single compartment
                                //% -- 2 for more than one
        abstemp,                //% Current Absolute Temperature
        V_in,                   //% Voltage In
        I_in,                   //% Current In
        Atotal,                 //% Membrane Total Area
        V_m,                    //% Membrane Voltage
        L,                      //% Neuron Cell length
        aj,                     //% Neuron Cell radius
        pi = 3.141592654;

/*****************************************************************/
/****                    Neuron Code Model ROUTINE                ***/

int neuron(ARGS)        /*      structure holding parms, inputs, outputs, etc.  */

{
I_in = (INPUT(a)*1e6);

if (INIT==TRUE) {       /*      First pass...allocate storage for previous value... */

/*  Pull in parameters from ifspec.ifs  */

        abstemp     = 273.15 + TEMPERATURE;
        aj          = PARAM(cell_radius);
        L           = PARAM(cell_length);
        compart_num     = PARAM(compartment_number);

        if (compart_num == 1){ /* Surface area of one cylinder, including ends  */

                Atotal = ((2*pi*aj*L*pow(100,2))+(2*pi*pow(aj,2)*pow(100,2)));
        }
        else{
                Atotal = (2*pi*aj*L*pow(100,2)); /*  Surface Area of cylinder for
                                                     multiple compartment*/
        }
        cm          = (PARAM(cap)*1e6)*Atotal;
```

```
        gNamax      = (PARAM(max_gna)*1e3)*Atotal;
        gKmax       = (PARAM(max_gk)*1e3)*Atotal;
        ciNa        = (PARAM(ci_na)*1e3);
        coNa        = (PARAM(co_na)*1e3);
        ciK         = (PARAM(ci_k)*1e3);
        coK         = (PARAM(co_k)*1e3);
        Vl          = (PARAM(v_l)*1e3);
        gL          = (PARAM(g_l)*1e3)*Atotal;
        V_r         = PARAM(v_rest);
        Q10         = PARAM(q10);
        if (V_r == 0){    /*  calculate resting potential if not defined in ifspec.ifs  */
              V_r = (((R*abstemp)/(Z*F))*log((coK + b*coNa)/(ciK +
                    b*ciNa)))*1.0E3;
        }

/* Allocate storage for static variables */

STATIC_VAR(previous_voltage) = (double *) malloc(sizeof(double));
        previous_voltage = STATIC_VAR(previous_voltage);

STATIC_VAR(previous_m) = (double *) malloc(sizeof(double));
        previous_m = STATIC_VAR(previous_m);

STATIC_VAR(previous_h) = (double *) malloc(sizeof(double));
        previous_h = STATIC_VAR(previous_h);

STATIC_VAR(previous_n) = (double *) malloc(sizeof(double));
        previous_n = STATIC_VAR(previous_n);

STATIC_VAR(previous_voltage_temp) = (double *) malloc(sizeof(double));
        previous_voltage_temp = STATIC_VAR(previous_voltage_temp);

STATIC_VAR(previous_m_temp) = (double *) malloc(sizeof(double));
        previous_m_temp = STATIC_VAR(previous_m_temp);

STATIC_VAR(previous_h_temp) = (double *) malloc(sizeof(double));
        previous_h_temp = STATIC_VAR(previous_h_temp);

STATIC_VAR(previous_n_temp) = (double *) malloc(sizeof(double));
        previous_n_temp = STATIC_VAR(previous_n_temp);

STATIC_VAR(lastT) = (double *) malloc(sizeof(double));
        lastT = STATIC_VAR(lastT);

STATIC_VAR(lastCurrent) = (double *) malloc(sizeof(double));
        lastCurrent = STATIC_VAR(lastCurrent);
  /* Set Nernst Potentials, VNa, and VK based on ionic concentrations */

VNa = (((R*abstemp)/(Z*F))*log(coNa/ciNa))*1e3;
VK  = (((R*abstemp)/(Z*F))*log(coK/ciK))*1e3;

/* Set M, N and H initial conditions... */
```

```
  ah = 0.07*exp(-0.05*(V_r+60));
  bh = 1/(1+exp(-0.1*(V_r+30)));

/*****************************************************************/
/***            Handle indeterminate cases when denominator = 0            ***/

if (V_r == -35){
        up = V_r + 1.0E-4;
        down = V_r - 1.0E-4;
        am = (-0.1*(up+35)/(exp(-0.1*(up+35))-1) + -0.1*(down+35)/
              (exp(-0.1*(down+35))-1))/2;
}
else {
        am = -0.1*(V_r+35)/(exp(-0.1*(V_r+35))-1);
}
        bm = 4.0*exp(-(V_r+60)/18);

if (V_r == -50){
        up = V_r + 1.0E-4;
        down = V_r - 1.0E-4;
        an = (-0.01*(up+50)/(exp(-0.1*(up+50))-1) + -0.01*(down+50)/
              (exp(-0.1*(down+50))-1))/2;
}
else {
        an = -0.01*(V_r+50)/(exp(-0.1*(V_r+50))-1);
}
        bn = 0.125*exp(-0.0125*(V_r+60));


/*****************************************************************/

MO = am/(am+bm);                        /*  Final Value Time Constants  */
HO = ah/(ah+bh);
NO = an/(an+bn);

/* Set previous_voltage value to zero... */

*previous_voltage    = V_r;
*previous_m          = MO;
*previous_h          = HO;
*previous_n          = NO;
*lastT         = 0;
*lastCurrent   = 0;
Vm             = V_r;
}
else {                                  /*  if INIT != true...not first pass    */

        if (T(0)==0.0){
                        *previous_voltage = V_r;
                        *previous_m = MO;
                        *previous_h = HO;
                        *previous_n = NO;
                        *lastT = 0;
```

```
                Vm = V_r;
                *lastCurrent = 0;
        }

        else {                  /*      if T(0) != 0.0      */
                previous_voltage = STATIC_VAR(previous_voltage);
                previous_m = STATIC_VAR(previous_m);
                previous_h = STATIC_VAR(previous_h);
                previous_n = STATIC_VAR(previous_n);
                previous_voltage_temp =
STATIC_VAR(previous_voltage_temp);
                previous_m_temp = STATIC_VAR(previous_m_temp);
                previous_h_temp = STATIC_VAR(previous_h_temp);
                previous_n_temp = STATIC_VAR(previous_n_temp);
                lastT     = STATIC_VAR(lastT);
                lastCurrent = STATIC_VAR(lastCurrent);
                if (T(1) != *lastT){
                        *previous_voltage = *previous_voltage_temp;
                        *previous_m = *previous_m_temp;
                        *previous_h = *previous_h_temp;
                        *previous_n = *previous_n_temp;

                        *lastT = T(1);

                        *lastCurrent = I_in/Atotal;
                }
                        else{
                        }
                        delta_T = T(0)-T(1);

                        ah = 0.07*exp(-0.05*(*previous_voltage+60));
                        bh = 1/(1+exp(-0.1*(*previous_voltage+30)));

/*****************************************************************/
/***         Handle indeterminate cases when denominator = 0          ***/

                        if (*previous_voltage == -35){
                                up = *previous_voltage + 1.0E-4;
                                down = *previous_voltage - 1.0E-4;
                                am = (-0.1*(up+35)/(exp(-0.1*(up+35))-1) +
                                        -0.1*(down+35)/(exp(-0.1*(down+35))-
                                        1))/2;
                        }
                        else {
                                am = -0.1*(*previous_voltage+35)/
                                    (exp(0.1*(*previous_voltage+35))-1);
                        }
                        bm = 4.0*exp(-(*previous_voltage+60)/18);

                        if (*previous_voltage == -50){
                                up = *previous_voltage + 1.0E-4;
                                down = *previous_voltage - 1.0E-4;
                                an = (-0.01*(up+50)/(exp(-0.1*(up+50))-1) +
```

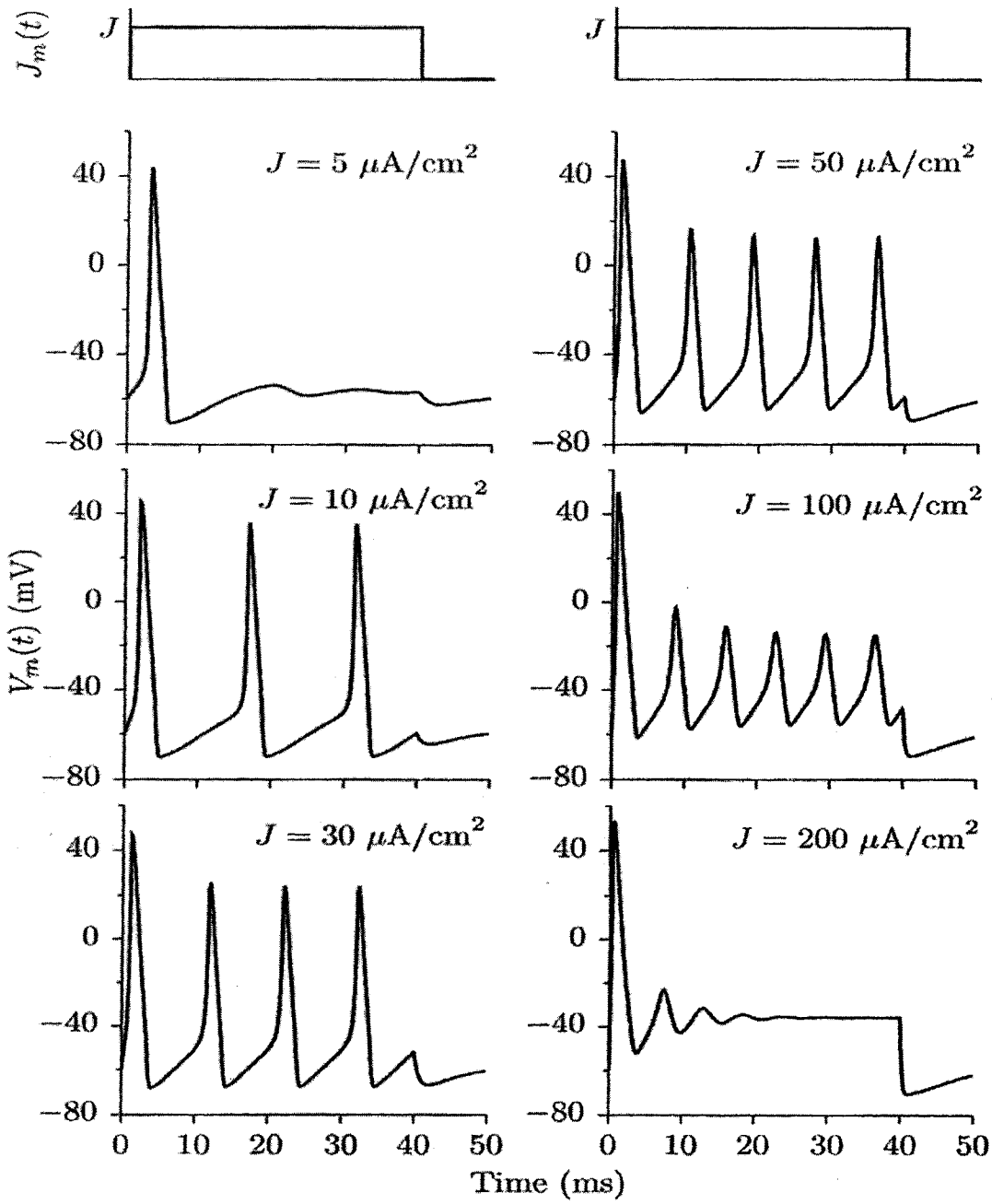
```
                                        -0.01*(down+50)/(exp(-0.1*(down+50))-
                                        1))/2;
                    }
                    else {
                                an = -0.01*(*previous_voltage+50)/
                                        (exp(-0.1*(*previous_voltage+50))-1);
                    }
/*************************************************************/
                    bn = 0.125*exp(-0.0125*(*previous_voltage+60));

/*  Calculate time constants        */

                    tauM = 1/(am+bm);
                    tauH = 1/(ah+bh);
                    tauN = 1/(an+bn);

/*   Calculate final M, H, & N values    */

                    infM = am/(am+bm);
                    infH = ah/(ah+bh);
                    infN = an/(an+bn);

/*   Solving m, h, &  n  by the Euler method */

                    m = *previous_m + ((((infM - *previous_m)/tauM) * delta_T)
                        * pow(Q10,((TEMPERATURE-6.3)/10))* 1e3);
                    h  = *previous_h + ((((infH - *previous_h)/tauH) * delta_T)
                        * pow(Q10,((TEMPERATURE-6.3)/10))* 1e3);
                    n  = *previous_n + ((((infN - *previous_n)/tauN) * delta_T)
                        * pow(Q10,((TEMPERATURE-6.3)/10))* 1e3);

/*  Calculating conductances using m, h, and n values calculated above   */

                    gNa =(gNamax*m*m*m*h);
                    gK = (gKmax*n*n*n*n);

/* Calculating ionic currents based in conductances, Nernst potentials, previous
voltages and cell capacitance values     */

                    IK = ((VK - *previous_voltage)*gK)/cm;
                    INa = ((VNa - *previous_voltage)*gNa)/cm;

                    Ileak = ((Vl - *previous_voltage)*gL)/cm;

/*  Calculating membrane current based on ionic currents   */

                    I_memb = IK + INa + Ileak + (*lastCurrent);

/*  Calculating new membrane voltage based on previous voltage and current
voltage for the current timestep using the Euler method  */

                    Vm = *previous_voltage + (I_memb*delta_T*1e3);
```

```
/* Store values for next iteration */
                    *previous_voltage_temp = Vm;
                    *previous_m_temp = m;
                    *previous_h_temp = h;
                    *previous_n_temp = n;
                }
        }

/*****************    Output to Port      ****************/

        OUTPUT(a) = Vm*1e-3;

return 0;

}
```

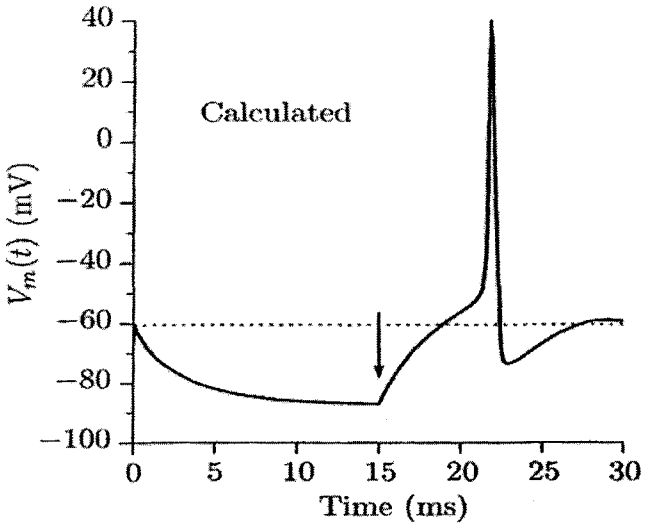
APPENDIX C

GRAPHS USED FOR VALIDATION

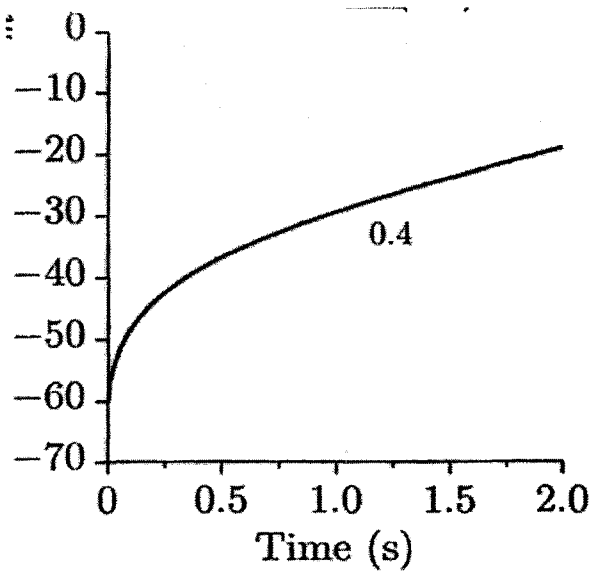
FROM WEISS [18]



Repetitive Activity,
Weiss, Page 232, Figure 4.60



Anode-Break Excitation
Weiss, Page 230, Figure 4.59



Accommodation
Weiss, Page 228, Figure 4.56

Bibliography

[1] Fromherz, P. and Stett, A., "Silicon-Neuron Junction: Capacitive Stimulation of an Individual Neuron on a Silicon Chip," *Physical Review Letters.*, vol. 75, no. 8, pp. 1670-1673, Aug.1995.

[2] Szlavik, R. B., "Strategies for improving neural signal detection using a neural-electronic interface," *IEEE Trans.Neural Syst.Rehabil.Eng*, vol. 11, no. 1, pp. 1-8, Mar.2003.

[3] Szlavik, R. B. and Jenkins, F. Varying the time delay of an action potential elicited with a neuralelectronic stimulator. Preceedings of the IEEE-EMBS Conference . 2004. San Fransico.

[4] Vassanelli, S. and Fromherz, P., "Neurons from rat brain coupled to transistors," *Applied Physics A*, vol. 65 pp. 85-88, 1997.

[5] Weis, R. and Fromherz, P., "Frequency dependent signal transfer in neuron transistors," *PhysicalReview E*, vol. 55, no. 1, pp. 877-889, Jan.1997.

[6] Szlavik, R. B. The impact of variations in membrane capacitance on the detected neural-electronic signal. Preceedings of the IEEE-EMBS Conference . 2002.

[7] Bove, M., Massobrio, G., Martinoia, S., and Grattarola, M., "Realistic simulations of neurons by means of an ad hoc modified version of SPICE," *Biol.Cybern.*, vol. 71, no. 2, pp. 137-145, 1994.

[8] Bunow, B., Segev, I., and Fleshman, J. W., "Modeling the electrical behavior of anatomically complex neurons using a network analysis program: excitable membrane," *Biol.Cybern.*, vol. 53, no. 1, pp. 41-56, 1985.

[9] Hodgkin, A. L. and Huxley, A. F., "A quantitative description of membrane current and its application to conduction and excitation in nerve," *J.Physiol*, vol. 117, no. 4, pp. 500-544, Aug.1952.

[10] Hodgkin, A. L. and Huxley, A. F., "The dual effect of membrane potential on sodium conductance in the giant axon of Loligo," *J.Physiol*, vol. 116, no. 4, pp. 497-506, Apr.1952.

[11] Hodgkin, A. L. and Huxley, A. F., "Currents carried by sodium and potassium ions through the membrane of the giant axon of Loligo," *J.Physiol*, vol. 116, no. 4, pp. 449-472, Apr.1952.

[12] Hodgkin, A. L. and Huxley, A. F., "The components of membrane conductance in the giant axon of Loligo," *J.Physiol*, vol. 116, no. 4, pp. 473-496, Apr.1952.

[13] Hodgkin, A. L., Huxley, A. F., and KATZ, B., "Measurement of current-voltage relations in the membrane of the giant axon of Loligo," *J.Physiol*, vol. 116, no. 4, pp. 424-448, Apr.1952.

[14] Segev, I., Fleshman, J. W., Miller, J. P., and Bunow, B., "Modeling the electrical behavior of anatomically complex neurons using a network analysis program: passive membrane," *Biol.Cybern.*, vol. 53, no. 1, pp. 27-40, 1985.

[15] Szlavik, R. B., Bhuiyan, A., Carver, A., and Jenkins, F., "Neural-electronic inhibition simulated with a neuron model implemented in spice," *IEEE Trans.Neural Syst.Rehabil.Eng*, 2006.

[16] Cox III, F. L., Kuhn, W. B., Murray, J. P., and Tynor, S. D., "Code-level modeling in XSPICE.," *Proceedings of the IEEE International Symposium on Circuits and Systems, 1992 (ISCAS '92)*, vol. 2 pp. 871-874, 1992.

[17] Hodgkin, A. L. and KATZ, B., "The effect of temperature on the electrical activity of the giant axon of the squid," *J.Physiol*, vol. 109, no. 1-2, pp. 240-249, Aug.1949.

[18] Weiss, T. F., *Cellular biophysics* Cambridge, Mass: MIT Press, 1996.

[19] Bhadra, N. and Kilgore, K. L., "Direct current electrical conduction block of peripheral nerve," *IEEE Trans.Neural Syst.Rehabil.Eng*, vol. 12, no. 3, pp. 313-324, Sept.2004.

[20] Matthews, G. G., *Cellular physiology of nerve and muscle*, 2nd ed. Boston: Blackwell Scientific Publications, 1991.

[21] Koch, C., *Biophysics of computation information processing in single neurons* New York: Oxford University Press, 1999.

[22] HAGIWARA, S. and OOMURA, Y., "The critical depolarization for the spike in the squid giant axon," *Jpn.J.Physiol*, vol. 8, no. 3, pp. 234-245, Sept.1958.

[23] Katz, B., *Electric Excitation of Nerve* London: Oxford University Press, 1939.

[24] Guttman, R. and Barnhill, R., "Temperature dependence of accommodation and excitation in space-clamped axons," *J.Gen.Physiol*, vol. 51, no. 6, pp. 759-769, June1968.

[25] Mauro, A., Conti, F., Dodge, F., and Schor, R., "Subthreshold behavior and phenomenological impedance of the squid giant axon," *J.Gen.Physiol*, vol. 55, no. 4, pp. 497-523, Apr.1970.

[26] Hodgkin, A. L. and Huxley, A. F., "A quantitative description of membrane current and its application to conduction and excitation in nerve," *J.Physiol*, vol. 117, no. 4, pp. 500-544, Aug.1952.

[27] Huxley, A. F., "Ion movements during nerve activity," *Ann.N.Y.Acad.Sci.*, vol. 81 pp. 221-246, Aug.1959.

[28] Berg-Johnsen, J. and Langmoen, I. A., "Temperature sensitivity of thin unmyelinated fibers in rat hippocampal cortex," *Brain Res.*, vol. 576, no. 2, pp. 319-321, Apr.1992.

[29] Chapman, R. A., "Dependence on temperature of the conduction velocity of the action potential of the squid giant axon," *Nature*, vol. 213, no. 81, pp. 1143-1144, Mar.1967.

[30] Jonas, P., "Temperature dependence of gating current in myelinated nerve fibers," *J.Membr.Biol.*, vol. 112, no. 3, pp. 277-289, Dec.1989.

[31] Kukita, F. and Yamagishi, S., "Excitation of squid giant axons below 0 degree C," *Biophys.J.*, vol. 35, no. 1, pp. 243-247, July1981.

[32] Rosenthal, J. J. and Bezanilla, F., "Seasonal variation in conduction velocity of action potentials in squid giant axons," *Biol.Bull.*, vol. 199, no. 2, pp. 135-143, Oct.2000.

[33] Nagel, L. and Rohrer, R., "Computer Analysis of Nonlinear Circuits Excluding Radiation (CANCER)," *IEEE Journal of Solid-State Circuits*, vol. SC–6 pp. 166-182, 1971.

[34] NGSPICE - Mixed Mode - Mixed Level Circuit Simulator. Website . 2006.

[35] Cox III, F. L., Kuhn, W. B., Li, H. W., Murray, J. P., Tynor, S. D., and Willis, M. J. XSPICE Software User's Manual. 1-124. 1992. Computer Science and Information Technology Laboratory, Georgia Tech Research Institute.

[36] Weiss, T. F., *Cellular biophysics* Cambridge, Mass: MIT Press, 1996.

[37] Rall, W., "Theoretical analysis of dendritic tree for input-output relation," in Reiss, R. F. (ed.) *Neural theory and modeling* Stanford, CA: Stanford University Press, 1964, pp. 73-97.

[38] Segev, I., Fleshman, J. W., and Burke, R., "Compartmental models of complex neurons," in Koch, C. and Segev, I. (eds.) *Methods in neuronal modeling. From synapses to network* Cambridge, MA: (Bradford Book) MIT Press, 1989, pp. 63-96.

[39] Cooley, J. W. and Dodge, F. A., Jr., "Digital computer solutions for excitation and propagation of the nerve impulse," *Biophys.J.*, vol. 6, no. 5, pp. 583-599, Sept.1966.

[40] Tai, C., de Groat, W. C., and Roppolo, J. R., "Simulation of nerve block by high-frequency sinusoidal electrical current based on the Hodgkin-Huxley model," *IEEE Trans.Neural Syst.Rehabil.Eng*, vol. 13, no. 3, pp. 415-422, Sept.2005.

[41] Tai, C., de Groat, W. C., and Roppolo, J. R., "Simulation analysis of conduction block in unmyelinated axons induced by high-frequency biphasic electrical currents," *IEEE Trans.Biomed.Eng*, vol. 52, no. 7, pp. 1323-1332, July2005.