

JAMES M. BOWER DAVID BEEMAN

The Book of GENESIS

Exploring Realistic Neural Models
with the GEneral NEural SImulation System



EXTRA
MATERIALS
extras.springer.com

INCLUDES
CD - ROM

SECOND EDITION

The Book of GENESIS

SECOND EDITION

JAMES M. BOWER DAVID BEEMAN

The Book of GENESIS

Exploring Realistic Neural Models
with the GEneral NEural SImulation System

SECOND EDITION

With 75 Illustrations



James M. Bower
Division of Biology
California Institute of Technology
Mail Stop 216-76
Pasadena, CA 91125
USA

David Beeman
Department of Electrical and
Computer Engineering
University of Colorado
Campus Box 425
Boulder, CO 80309-0425
USA

Publisher: Allan M. Wylde
Publishing Associate: Keisha Sherbecoe
Marketing Manager: Kenneth Quinn
Production Manager: Anthony K. Guardiola
Manufacturing Supervisor: Jeffrey Taub
Cover Artist: Erika Oller

Library of Congress Cataloging-in-Publication Data
Bower, James M.

The book of GENESIS: exploring realistic neural models with the
GEneral NEural SImulation System / by James M. Bower, David Beeman. —
2nd ed.

p. cm.

Includes bibliographical references and index.

ISBN 0-387-94938-0 (alk. paper)

Additional material to this book can be downloaded from <http://extra.springer.com>.

1. Neural networks (Computer science). 2. Computer simulation.
3. GENESIS (Computer file). I. Beeman, David, 1938— . II. Title.
QA76.87.B69 1997
006.3'—dc21

97-33270

Printed on acid-free paper.

©1998, 1995 Springer-Verlag New York, Inc.
Published by TELOS®, The Electronic Library of Science, Santa Clara, California.
TELOS® is an imprint of Springer-Verlag New York, Inc.

This work consists of a printed book and a CD-ROM packaged with the book, both of which are protected by federal copyright law and international treaty. The book may not be translated or copied in whole or in part without the permission of the publisher (Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010, USA) except for brief excerpts in connection with reviews or scholarly analysis. For copyright information regarding the CD-ROM, please consult the printed information packaged with the CD-ROM in the back of this publication, which is also stored as a "readme" file on the CD-ROM. Use of this work in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed other than those expressly granted in the CD-ROM copyright and disclaimer information is forbidden.

The use of general descriptive names, trade names, trademarks, etc., in this publication, even if the former are not especially identified, is not to be taken as a sign that such names, as understood by the Trade Marks and Merchandise Marks Act, may accordingly be used freely by anyone. Where those designations appear in this book and Springer-Verlag was aware of a trademark claim, the designations follow the capitalization style used by the manufacturer.

Photocomposed copy prepared using the authors' LaTeX files.

9 8 7 6 5 4 3 2 1

ISBN 0-387-94938-0 Springer-Verlag New York Berlin Heidelberg SPIN 10556516



TELOS, The Electronic Library of Science, is an imprint of Springer-Verlag New York with publishing facilities in Santa Clara, California. Its publishing program encompasses the natural and physical sciences, computer science, economics, mathematics, and engineering. All TELOS publications have a computational orientation to them, as TELOS' primary publishing strategy is to wed the traditional print medium with the emerging new electronic media in order to provide the reader with a truly interactive multimedia information environment. To achieve this, every TELOS publication delivered on paper has an associated electronic component. This can take the form of book/diskette combinations, book/CD-ROM packages, books delivered via networks, electronic journals, newsletters, plus a multitude of other exciting possibilities. Since TELOS is not committed to any one technology, any delivery medium can be considered.

The range of TELOS publications extends from research level reference works through textbook materials for the higher education audience, practical handbooks for working professionals, as well as more broadly accessible science, computer science, and high technology trade publications. Many TELOS publications are interdisciplinary in nature, and most are targeted for the individual buyer, which dictates that TELOS publications be priced accordingly.

Of the numerous definitions of the Greek word "telos," the one most representative of our publishing philosophy is "to turn," or "turning point." We perceive the establishment of the TELOS publishing program to be a significant step towards attaining a new plateau of high quality information packaging and dissemination in the interactive learning environment of the future. TELOS welcomes you to join us in the exploration and development of this frontier as a reader and user, an author, editor, consultant, strategic partner, or in whatever other capacity might be appropriate.

TELOS, The Electronic Library of Science
Springer-Verlag Publishers
3600 Pruneridge Avenue, Suite 200
Santa Clara, CA 95051



TELOS Diskettes

Unless otherwise designated, computer diskettes packaged with TELOS publications are 3.5" high-density DOS-formatted diskettes. They may be read by any IBM-compatible computer running DOS or Windows. They may also be read by computers running NEXTSTEP, by most UNIX machines, and by Macintosh computers using a file exchange utility.

In those cases where the diskettes require the availability of specific software programs in order to run them, or to take full advantage of their capabilities, then the specific requirements regarding these software packages will be indicated.

TELOS CD-ROM Discs

For buyers of TELOS publications containing CD-ROM discs, or in those cases where the product is a stand-alone CD-ROM, it is always indicated on which specific platform, or platforms, the disc is designed to run. For example, Macintosh only; Windows only; cross-platform, and so forth.

TELOSpub.com (Online)

Interact with TELOS online via the Internet by setting your World-Wide-Web browser to the URL: <http://www.telospub.com>.

The TELOS Web site features new product information and updates, an online catalog and ordering, samples from our publications, information about TELOS, data-files related to and enhancements of our products, and a broad selection of other unique features. Presented in hypertext format with rich graphics, it's your best way to discover what's new at TELOS.

TELOS also maintains these additional Internet resources:

*gopher://gopher.telospub.com
ftp://ftp.telospub.com*

For up-to-date information regarding TELOS online services, send the one-line e-mail message:

send info to: info@TELOSpub.com.

Preface

This is the second edition of a step-by-step tutorial for professionals, researchers and students working in the area of neuroscience in general, and computational neuroscience in particular. It can also be used as an interactive self-study guide to understanding biological neuronal and network structure for those working in the area of artificial neural networks and the cognitive sciences. The tutorials are based upon the GENESIS neural simulation system, which is now being used for teaching and research in at least 26 countries. The following chapters consist of a combination of edited contributions from researchers in computational neuroscience and current users of the system, as well as several chapters that we have written ourselves.

This book, and the tutorial simulations on which it is based, grew out of a simulation laboratory accompanying the annual Methods in Computational Neuroscience course taught at the Marine Biological Laboratory in Woods Hole, MA from 1988 to 1992. Since that time, the tutorials have been further developed and refined while being used in courses taught at Caltech and several other institutions, including the Crete course in Computational Neuroscience. For this second edition, we have made many revisions and additions based on comments, suggestions and corrections from members of the GENESIS Users Group, BABEL, and from students and teachers who have used this book.

The release of GENESIS version 2.1, with its many new features, has resulted in the addition of two new chapters and the addition of new material to existing chapters. With the availability of the GENESIS chemical kinetics modeling library and its graphical interface (*Kinetikit*), we expect to see GENESIS being increasingly used to relate cellular and network properties to biochemical signaling pathways (Chapter 10). The increasing sophistication, realism and size of large network models and the use of computationally intensive automatic parameter fitting methods have led to the development of a new parallel version of GENESIS (PGENESIS) which allows GENESIS to run on parallel computers and networks of workstations (Chapter 21). Other new additions include a section describing ways to implement synaptic modification (learning), a section describing uses of a new type

of GENESIS ionic channel (the two-dimensional tabulated channel) which allows modeling of a wide variety of voltage and ionic concentration-dependent channels, a description of improvements in the procedure for implementing fast “implicit” numerical methods in GENESIS simulations, and descriptions of many new GENESIS commands and simulation components.

This new edition is accompanied by a CD-ROM containing the complete GENESIS 2.1 distribution with *Kinetikit*, PGENESIS and hypertext reference manuals. It also includes numerous tutorial simulations and example simulation scripts, including all of those used in the book, plus a collection of GENESIS models taken from the BABEL archives. Please see Appendix A for further details.

Part I of the book teaches concepts in neuroscience and neural modeling by means of interactive computer tutorials. These allow the student to perform realistic simulations and experiments on model neural systems. The simulations are user-friendly with on-line help and may be used without any prior knowledge of the GENESIS simulator or computer programming. The tutorials in Part II teach the use and programming of the GENESIS simulator for the construction of one’s own simulations. Each tutorial is accompanied by a number of suggested exercises, “experiments,” or projects which may be either assigned as homework or used for self-study.

Although they form a sequence, these tutorials were designed so that they may be used independently, as supplemental material for existing courses. This presentation, as well as the variety of suggested exercises and optional projects, makes it possible to explore topics at various levels of depth. For example, a survey course without much time to devote to a simulation laboratory might use only one or two simulations selected from the first few chapters, which treat the Hodgkin–Huxley model, passive propagation in dendrites, and the temporal summation of synaptic potentials in a multi-compartmental neuron model. A more advanced course, or one that emphasizes the use of computer simulation as a tool for the understanding of the nervous system, would place more emphasis on the later chapters that deal with the role of ionic conductances in burst firing of neurons, central pattern generator circuits, cortical networks and modeling of biochemical signaling pathways. A course that can devote time to student simulation projects can use the tutorials in Part II to quickly get students to the point of creating their own simulations. In the introductory chapter (Table 1.1), we list the correspondence between the Part I chapters and those in several popular neuroscience textbooks. Later, in Sec. 3.2, we give an overview of the tutorials used in this book.

We have found that even experienced researchers who are familiar with the concepts presented in the introductory chapters have been able to learn something from the accompanying simulations, and have been able to use the more advanced simulations as a starting point for original research simulations. As interest in the field of computational neuroscience continues to expand, and as increasing numbers of neuroscientists recognize the necessary connection between modeling and experimental neurobiology, we hope and an-

ticipate that this book will promote access to the power of realistic simulations of neural structures.

Acknowledgments

In writing this book, we have benefited from the help of many people. Most of the credit is due to the hard work of the contributors (listed on page xxiii) who unselfishly devoted their time to this project. We, of course, take full responsibility for the inevitable errors and omissions. The content and presentation have greatly benefited from the comments and suggestions of those from the many institutions that are now using GENESIS in teaching. Particular thanks are due to Jacques Brisson (Ecole des Hautes Etudes Commerciales), Mark Kamath (McMaster University), Allen Plummer (University of Nevada, Reno) and Ed Vigmond (University of Toronto), who have reviewed or used preliminary drafts of the book. We are indebted to all the students who helped, sometimes under adverse conditions, to refine and debug these simulations during the Woods Hole course. We have also benefited from helpful suggestions by Upi Bhalla (National Centre for Biological Sciences, Bangalore), Erik De Schutter (University of Antwerp), Jason Leigh (University of Illinois, Chicago) and Eve Marder (Brandeis), who have reviewed drafts of chapters from the book, and from Carolyn Keaton (Tennessee State University), who reviewed several chapters and tested the accompanying exercises. We also appreciate the help of Chris Assad (Caltech) in generating figures for Chapter 9.

Special thanks go to our publisher, Allan Wylde of TELOS, for his encouragement and his patience during the long process of completing this book. The whimsical artwork on the cover is the creation of Erika Oller, who is becoming increasingly famous for her unique illustrations.

This book and the accompanying educational tutorials could not have been developed without the financial support of the Division of Biological Instrumentation and Resources of the National Science Foundation. We wish to thank them for recognizing and supporting the important and growing synergy between research, education and computational science.

Above all, thanks are due to the numerous GENESIS developers who have contributed and continue to contribute to the development of this system: this especially includes Matt Wilson and Upinder Bhalla, the original creators of the simulator and its interface; Erik De Schutter, Alex Protopapas, Mike Vanier, and Upi again, who continue to contribute both ideas and energy to the expansion of the system; Dave Bilitch, Venkat Jagadish, and Jenny Forss for their tireless software support, and former students in our laboratory at Caltech, Dieter Jaeger, Maurice Lee, Maneesh Sahani and Michael Speight for ideas, creativity and guidance to this project.

GENESIS users from outside Caltech who have also played a particularly important role in the development of GENESIS include: David Berkowicz (Yale), Bill Broadley (Univer-

sity of Pittsburgh), Kevin Cunningham (MIT), Dale Fay (ERIM), Nigel Goddard (Pittsburgh Supercomputer Center), Randy Gobbel (UC, San Diego), Bruce Graham (Australian National University), Mike Hasselmo (Harvard), William Holmes (Ohio University), Rich Murphey (University of Texas, Galveston), Mark Nelson (University of Illinois), Jim Olds (NIH), Walter Schneider (University of Pittsburgh), David Senseman (University of Texas, San Antonio), Diana Smetters (Salk Institute) and Mike Walker (Brown), as well as many other members of BABEL, who have contributed to GENESIS development through documentation, simulations, simulator code, bug fixes, ports to other machines, suggestions and provocative questions (and sometimes criticisms).

Finally, we are most grateful for the patience and understanding of our families, friends and pets, who have suffered much neglect during the first writing of this book, and have now suffered yet again.

Pasadena, California
Boulder, Colorado

James M. Bower
David Beeman

Contents

Preface	vii
About the Authors	xxi
Contributors	xxiii
I Neurobiological Tutorials with GENESIS	1
1 Introduction	3
1.1 Computational Neuroscience	3
1.2 Using This Book	4
2 Compartmental Modeling	7
2.1 Modeling Neurons	7
2.1.1 Detailed Compartmental Models	8
2.1.2 Equivalent Cylinder Models	9
2.1.3 Single and Few Compartment Models	10
2.2 Equivalent Circuit of a Single Compartment	10
2.3 Axonal Connections, Synapses and Networks	12
2.4 Simulation Accuracy	13
2.4.1 Choice of Numerical Integration Technique	13
2.4.2 Integration Time Step	14
2.4.3 Accuracy of GENESIS	15
3 Neural Modeling with GENESIS	17
3.1 What Is GENESIS?	17

3.1.1	Why Use a General Simulator?	17
3.1.2	GENESIS Design Features	18
3.1.3	GENESIS Development	20
3.2	Introduction to the Tutorials	20
3.3	Introduction to the GENESIS Graphical Interface	22
3.3.1	Starting the Simulation	22
3.3.2	The Control Panel	23
3.3.3	Using Help Menus	24
3.3.4	Displaying the Simulation Results	26
4	The Hodgkin–Huxley Model	29
4.1	Introduction	29
4.2	Historical Background	30
4.3	The Mathematical Model	34
4.3.1	Electrical Equivalent Circuit	34
4.3.2	HH Conventions	35
4.3.3	The Ionic Current	36
4.4	Voltage Clamp Experiments	38
4.4.1	Characterizing the K Conductance	39
4.5	GENESIS: Voltage Clamp Experiments	41
4.6	Parameterizing the Rate Constants	44
4.7	Inactivation of the Na Conductance	45
4.8	Current Injection Experiments	47
4.9	Exercises	47
5	Cable and Compartmental Models of Dendritic Trees	51
5.1	Introduction	51
5.2	Background	53
5.2.1	Dendritic Trees: Anatomy, Physiology and Synaptology	53
5.2.2	Summary	55
5.3	The One-Dimensional Cable Equation	56
5.3.1	Basic Concepts and Assumptions	56
5.3.2	The Cable Equation	56
5.4	Solution of the Cable Equation for Several Cases	59
5.4.1	Steady-State Voltage Attenuation with Distance	59
5.4.2	Voltage Decay with Time	60
5.4.3	Functional Significance of λ and τ_m	61
5.4.4	The Input Resistance R_{in} and “Trees Equivalent to a Cylinder”	62
5.4.5	Summary of Main Results from the Cable Equation	64
5.5	Compartmental Modeling Approach	65

5.6	Compartmental Modeling Experiments	68
5.7	Main Insights for Passive Dendrites with Synapses	71
5.8	Biophysics of Excitable Dendrites	73
5.9	Computational Function of Dendrites	75
5.10	Exercises	75
6	Temporal Interactions Between Postsynaptic Potentials	79
6.1	Introduction	79
6.2	Electrical Model of a Patch of Membrane	80
6.2.1	Voltage Response of Passive Membrane to a Current Pulse	81
6.3	Response to Activation of Synaptic Channels	85
6.3.1	The Postsynaptic Current	85
6.3.2	The Postsynaptic Potential	86
6.3.3	Smooth Synaptic Conductance Change: The “Alpha Function”	88
6.4	A Remark on Synaptic Excitation and Inhibition	89
6.5	GENESIS Experiments with PSPs	90
6.5.1	Temporal Summation of Postsynaptic Potentials	91
6.5.2	Nonlinear Summation of Postsynaptic Potentials	93
6.6	Concluding Remarks	94
6.7	Exercises and Projects	95
7	Ion Channels in Bursting Neurons	97
7.1	Introduction	97
7.2	General Properties of Molluscan Neurons	99
7.3	Ionic Conductances — The Dance of the Ions	101
7.3.1	Action Potential Related Conductances	102
7.3.2	Control of Bursting Properties	106
7.4	A Model Molluscan Neuron	112
7.4.1	Adrift in Parameter Space	112
7.4.2	Implementation of the Model	114
7.4.3	Modeling the Channels	115
7.5	The Molluscan Neuron Simulation	118
7.5.1	Using Neurokit	118
7.5.2	Understanding the Results	119
7.6	The Traub Model CA3 Pyramidal Cell	121
7.6.1	Experiments with the Traub Model	122
7.6.2	Firing Patterns	126
7.7	Exercises	127

8	Central Pattern Generators	131
8.1	Introduction	131
8.2	Two-Neuron Oscillators	134
8.2.1	Phase Equation Model of Coupled Oscillators	134
8.2.2	Simulation Parameters	136
8.2.3	Initial Conditions	137
8.2.4	Synaptic Coupling	138
8.2.5	Non-Phase Equation Models	139
8.3	Four-Neuron Oscillators	140
8.3.1	Chains of Coupled Oscillators	140
8.3.2	Simulation Parameters	142
8.3.3	Modeling Gaits	144
8.4	Summary	146
8.5	Exercises	146
9	Dynamics of Cerebral Cortical Networks	149
9.1	Introduction	149
9.2	Piriform Cortex	150
9.3	Structure of the Model	150
9.3.1	Cellular Complexity	151
9.3.2	Network Circuitry	152
9.4	Electroencephalography	154
9.5	Using the Tutorial	158
9.5.1	Getting Started	158
9.5.2	Generating Simulated Data	159
9.5.3	Initial Look at Simulated Activity	160
9.5.4	Observing Network Behavior	162
9.5.5	Varying Network Parameters	162
9.6	Detailed Examination of Network Behavior	164
9.7	Summary	167
9.8	Exercises	167
10	The Network Within: Signaling Pathways	169
10.1	Introduction	169
10.1.1	Nomenclature	170
10.1.2	A Short Short Course in Biochemistry	171
10.1.3	Common Signaling Pathways	172
10.2	Modeling Signaling Pathways	175
10.2.1	Theory	175
10.2.2	Sources of Data	176

10.2.3	Figuring Out the Mechanisms	176
10.2.4	Reaction Rate Constants	178
10.2.5	Enzyme Rate Constants	178
10.2.6	Initial Concentrations	179
10.2.7	Refining the Model	180
10.3	Building Kinetics Models with GENESIS and <i>Kinetikit</i>	180
10.3.1	The kinetics Library	180
10.3.2	Kinetikit	181
10.3.3	A Feedback Model	184
10.3.4	Beyond Kinetikit	187
10.3.5	Connecting Kinetic Models to the Rest of GENESIS	188
10.4	Summary: Molecular Computation	189
10.5	Exercises	190
II	Creating Simulations with GENESIS	193
11	Constructing New Models	195
11.1	Structurally Realistic Modeling	196
11.2	The Modeling Process	198
11.2.1	Single Neurons or Networks	199
11.2.2	Modeling Steps	200
12	Introduction to GENESIS Programming	203
12.1	Simulating a Simple Compartment	203
12.2	Getting Started with GENESIS	203
12.3	GENESIS Objects and Elements	205
12.3.1	Creating and Deleting Elements	206
12.3.2	Examining and Modifying Elements	206
12.4	Running a GENESIS Simulation	208
12.4.1	Adding Graphics	208
12.4.2	Linking Elements with Messages	209
12.4.3	Adding Buttons to a Form	210
12.5	How GENESIS Performs a Simulation	211
12.6	Exercises	212
13	Simulating a Neuron Soma	215
13.1	Some GENESIS Script Language Conventions	215
13.1.1	Defining Functions in GENESIS	215
13.2	Making a More Realistic Soma Compartment	218

13.2.1	Some Remarks on Units	218
13.2.2	Building a “Squid-Like” Soma	219
13.2.3	GIGO (Garbage In, Garbage Out)	221
13.3	Debugging GENESIS Scripts	223
13.4	Exercises	224
14	Adding Voltage-Activated Channels	225
14.1	Review	225
14.2	More Fun With XODUS	226
14.3	Voltage-Activated Channel Objects	229
14.3.1	The hh_channel Object	230
14.3.2	Adding Hodgkin–Huxley Na and K Channels to the Soma	231
14.4	Final Additions and Improvements	233
14.4.1	Use of the Compartment <i>initVm</i> Field	234
14.4.2	Overlaying GENESIS Plots	235
14.5	Extended Objects	236
14.6	Exercises	240
15	Adding Dendrites and Synapses	243
15.1	Adding a Dendrite Compartment	243
15.2	Providing Synaptic Input	246
15.3	Connections Between Neurons	248
15.4	Learning and Synaptic Plasticity	251
15.4.1	Continous Modification of the Synaptic Weight	251
15.4.2	Use of the MOD Message	251
15.4.3	Hebbian Learning with the hebbsynchan	251
15.4.4	Customizing the synchan or hebbsynchan	252
15.5	Where Do We Go from Here?	252
15.6	Exercises	253
16	Automating Cell Construction with the Cell Reader	255
16.1	Introduction	255
16.2	Creating a Library of Prototype Elements	256
16.2.1	Future Changes in the Cell Reader	256
16.2.2	The <i>protodefs.g</i> Script	257
16.3	The Format of the Cell Descriptor File	259
16.4	Modifying the Main Script to Use the Cell Reader	262
16.5	The Neurokit Simulation	262
16.6	Exercises	263

17 Building a Cell with Neurokit	265
17.1 Introduction and Review	265
17.2 Customizing the <i>userprefs</i> File	266
17.2.1 Step 1	267
17.2.2 Step 2	267
17.2.3 Step 3	270
17.3 The Cell Descriptor File	273
17.4 Some Experiments Using Neurokit	273
17.5 Exercises and Projects	276
18 Constructing Neural Circuits and Networks	279
18.1 Introduction	279
18.2 The <i>Orient.tut</i> Simulation	280
18.3 Running the Simulation	280
18.4 Creating a Network Simulation	282
18.5 Defining Prototypes	282
18.6 Creating Arrays of Cells	284
18.7 Making Synaptic Connections	286
18.7.1 Specifying Individual Synaptic Connections	287
18.7.2 Commands Involving Groups of Synapses	288
18.7.3 Utility Functions for Synapses	297
18.8 Setting Up the Inputs	299
18.9 Summary	300
18.10 Exercises	300
19 Implementing Other Types of Channels	301
19.1 Introduction	301
19.2 Using Experimental Data to Make a tabchannel	302
19.2.1 Setting the tabchannel Internal Fields	304
19.2.2 Testing and Editing the Channel	307
19.3 Using Equations for the Rate Constants	311
19.4 Implementing Calcium-Dependent Conductances	314
19.4.1 Calculating the Calcium Concentration	314
19.4.2 The AHP Current	317
19.4.3 The C-Current	318
19.4.4 Other Uses of the table Object	320
19.4.5 The vdep_gate Object	321
19.4.6 Using the tab2Dchannel Object	321
19.5 NMDA Channels	324
19.6 Gap Junctions	325

19.7	Dendrodendritic Synapses	326
19.8	Exercises	327
20	Speeding Up GENESIS Simulations	329
20.1	Introduction	329
20.2	Some General Hints	329
20.3	Numerical Methods Used in GENESIS	332
20.3.1	The Differential Equations Used in GENESIS	332
20.3.2	Explicit Methods	333
20.3.3	Implicit Methods	334
20.3.4	Instability and Stiffness	335
20.3.5	Implementation of the Implicit Methods	337
20.4	The <i>setmethod</i> Command	337
20.5	Using the hsolve Object	338
20.5.1	Modes of Operation	339
20.5.2	Rules for Table Dimensions	343
20.6	Setting up hsolve	343
20.6.1	The <i>findsolvefield</i> Function	345
20.6.2	The DUPLICATE Action	346
20.7	Experiments with the hsolve Object	346
20.8	Exercises	347
21	Large-Scale Simulation Using Parallel GENESIS	349
21.1	Introduction	349
21.2	Classes of Parallel Platforms	351
21.3	Parallel Script Development	352
21.4	Script Language Programming Model	353
21.4.1	Parallel Virtual Machine	353
21.4.2	Namespace	353
21.4.3	Execution (Threads and Synchronization)	354
21.4.4	Simulation and Scheduling	355
21.4.5	Node-Specific Script Processing	355
21.4.6	Asynchronous Simulation	356
21.4.7	Zones and Node Identifiers	356
21.4.8	Remote Function Call	357
21.4.9	Asynchronous Remote Function Call	358
21.4.10	Message Creation	360
21.5	Running PGENESIS	360
21.5.1	The <i>pgenesis</i> Startup Script	361
21.5.2	Debug Modes	362

21.6	Network Model Example	363
21.6.1	Setup	363
21.6.2	Simulation Control	366
21.6.3	Lookahead	367
21.7	Parameter Search Examples	368
21.8	I/O Issues	374
21.9	Summary of Script Language Extensions	375
21.9.1	Startup/Shutdown	375
21.9.2	Adding Messages	376
21.9.3	Synaptic Connections	376
21.9.4	Remote Command Execution and Synchronization	376
21.9.5	PGENESIS Objects	376
21.9.6	Modifiable PGENESIS Parameters	377
21.9.7	Unsupported and Dangerous Operations	377
21.10	Exercises	378
22	Advanced XODUS Techniques: Simulation Visualization	381
22.1	Introduction	381
22.2	What Can Your User Interface Do for You?	382
22.3	Draw/Pix Philosophy	382
22.4	Meet the Cast	384
22.4.1	The Draw Widget Family	384
22.4.2	The Pix Family	387
22.5	XODUS Events	394
22.5.1	Returning Arguments to Script Functions	395
22.6	Using Advanced Widgets: A Network Builder	396
22.6.1	The Library Window	396
22.6.2	Making Prototype Cells	397
22.6.3	The Work Window	398
22.6.4	Editing Cells	399
22.6.5	Connecting Cells	400
22.6.6	Plotting Cell Activity	401
22.6.7	Running Netkit	402
22.6.8	Extending Netkit	403
22.7	Interface vs. Simulation	404
22.8	Summary	405

A Acquiring and Installing GENESIS	407
A.1 System Requirements	407
A.2 Using the CD-ROM	407
A.3 Obtaining GENESIS over the Internet	408
A.4 Installation and Documentation	408
A.5 Copyright Notice	409
B GENESIS Script Listings	411
B.1 tutorial2.g	411
B.2 tutorial3.g	413
B.3 tutorial4.g	415
B.4 tutorial5.g	420
B.5 hhchan.g	422
B.6 hhchan_K.g	425
B.7 userprefs.g	426
B.8 cellproto.g	428
Bibliography	431
Index of GENESIS Commands	449
Index of GENESIS Objects	451
Index	453

About the Authors

The main authors/editors of The Book of GENESIS are James M. Bower and David Beeman. Other contributors to the book are Upinder S. Bhalla, Avis H. Cohen, Sharon Crook, Erik De Schutter, Nigel H. Goddard, Greg Hood, Mark Nelson, Alexander Protopapas, John Rinzel, Idan Segev, Michael Vanier and Matthew A. Wilson.

Dr. Bower is Associate Professor of Biology at the California Institute of Technology in Pasadena, California. One of the founders of the Computation and Neural Systems Graduate Program at Caltech, he has been actively involved in establishing the field of Computational Neuroscience. He is a founder and managing editor of the Journal of Computational Neuroscience, and principal organizer of the Annual International Meeting in Computational Neuroscience. Research in his laboratory is focused on anatomical, physiological, and model based studies of information processing in cortical structures of the mammalian brain. He received his Ph.D. in Neurophysiology from the University of Wisconsin, Madison in 1981.

Dr. Beeman is Professor Adjunct of Electrical and Computer Engineering at the University of Colorado, Boulder where he is developing educational materials for computational neuroscience, using the GENESIS simulator. He spent 20 years at Harvey Mudd College engaged in undergraduate teaching and research in computational solid state physics after receiving his Ph.D. in theoretical solid state physics from UCLA in 1967.

Contributors

David Beeman (Chapters 1–3, 7 and 12–20), Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309-0425

Upinder S. Bhalla (Chapters 10 and 21), National Centre for Biological Sciences, TIFR Centre, IISc Campus, PO Box 1234, Bangalore 560012, India

James M. Bower (Chapters 1–3, 7, 9 and 11), Division of Biology 216-76, California Institute of Technology, Pasadena, CA 91125

Avis H. Cohen (Chapter 8), Center for Neural and Cognitive Sciences, University of Maryland, College Park, MD 20742

Sharon Crook (Chapter 8), Center for Computational Biology, PO Box 173505, Montana State University, Bozeman, MT 59717-3505

Erik De Schutter (Chapter 20), Born-Bunge Foundation, University of Antwerp - UIA, Universiteitsplein 1, B2610 Antwerp, Belgium

Nigel H. Goddard (Chapter 21), Pittsburgh Supercomputing Center, Pittsburgh, PA 15213

Greg Hood (Chapter 21), Pittsburgh Supercomputing Center, Pittsburgh, PA 15213

Mark Nelson (Chapter 4), Beckman Institute, University of Illinois, Urbana, IL 61801

Alexander Protopapas (Chapter 9), Division of Biology 216-76, California Institute of Technology, Pasadena, CA 91125

John Rinzel (Chapter 4), Mathematical Research Branch, National Institutes of Health, Bethesda, MD 20814

Idan Segev (Chapters 5 and 6), Department of Neurobiology, Institute of Life Sciences, Hebrew University, Jerusalem 91904, Israel

Michael Vanier (Chapter 18), Division of Biology 216-76, California Institute of Technology, Pasadena, CA 91125

Matthew A. Wilson (Chapter 12), Department of Brain and Cognitive Sciences, Massachusetts Institute of Technology, Cambridge, MA 02139

Part I

Neurobiological Tutorials with GENESIS

Chapter 1

Introduction

JAMES M. BOWER and DAVID BEEMAN

1.1 Computational Neuroscience

The last several years have seen a dramatic increase in the number of neurobiologists building or using computer-based models as a regular part of their efforts to understand how different neural systems function (Eeckman and Bower 1993, Bower 1992). As experimental data continue to be amassed, it is increasingly clear that detailed physiological and anatomical data alone are not enough to infer how neural circuits work. Experimentalists appear to be recognizing the need for the quantitative approach to exploring the functional consequences of particular neuronal features that is provided by modeling. This combination of modeling and experimental work has led to the creation of the new discipline of computational neuroscience (Eeckman and Bower 1993).

More than the use of models *per se*, we believe that computational neuroscience is most distinguished from classical neurobiology by an explicit focus on how the nervous system computes. Thus, instead of obtaining experimental information about the structure of the nervous system for its own sake, a computational approach involves collecting that information most relevant at the moment for the advancement of functional understanding. In our hands, models, especially those based on the detailed physiology and anatomy of the brain region in question, capture what is known about this region while also directing further experimental investigations. These same models can then provide an interpretation for the data that were obtained. Thus, the interaction between experiments and computer modeling is increasingly iterative and interdependent.

This approach to the interaction between computer models and experimental investigations obviously goes far beyond the traditional notion that experiments are simply for “testing” theoretical ideas. Furthermore, we believe that it requires that models not primarily be designed to cast novel ideas in more or less biological detail. Instead, modeling becomes a mechanism for generating new ideas based on the anatomy and physiology of the circuits themselves (Bower 1995). This issue is discussed in more detail in Chapter 11. However, a major objective of this tutorial guide to the use of the GENESIS simulator, and of GENESIS itself, is to provide future computational neurobiologists with the tools to construct models of this sort.

1.2 Using This Book

This book is divided into two parts that are intended to serve two different purposes. Part I describes the use of eight interactive tutorials which introduce modeling at different scales, ranging from parts of neurons to large neuronal networks. Depending on the audience, these tutorials can serve several purposes. The tutorial chapters are written so that they can be used by students of neurobiology, as well as by engineers, physicists, computer scientists, and others who are interested in increasing their knowledge of the nervous system. Each chapter begins by presenting the necessary theoretical background for the topic and then uses the tutorial simulations to further explore the implications of the theory. The book is designed to be used either for self-study, or for use in a course as a supplement to an introductory neuroscience text at the level of *Neurobiology* (Shepherd 1994), *From Neuron to Brain* (Nicholls, Martin and Wallace 1992) or *Principles of Neural Science* (Kandel, Schwartz and Jessell 1991). For a course that stresses a more quantitative approach at the single cell level, *Foundations of Cellular Neurophysiology* (Johnston and Wu 1995) would be an appropriate companion to *The Book of GENESIS*. Table 1.1 lists the chapters in these texts that correspond to the tutorials in Part I. Later, in Sec. 3.2, we present an overview of these tutorials.

In addition to exploring basic neurobiological principles, these chapters also introduce general concepts relevant to the process of modeling itself. Although no knowledge of computer programming is needed to use the tutorials, one may nevertheless learn a great deal about computer modeling from these simulations. For example, we consider the effect of numerical integration time steps on the results of simulations. We gain some understanding of why slight changes in some parameters can greatly affect the results, whereas changes in others have little effect. A recurring theme is the question, “When can I get by with a simple model, and when must I capture all the details in my model?” When constructing a model, we rarely have all the experimental information that we would like. Several of these chapters address the issue of using modeling as a means of bridging the gap between experiment and theory.

<i>BoG</i>	<i>Shepherd</i>	<i>Nicholls et al.</i>	<i>Kandel et al.</i>	<i>Johnston and Wu</i>
4	4, 5	4	5,6,8	3,6
5	—	5	7	4
6	7	7	10,11	13
7	25	4,13	—	7
8	20	15	—	—
9	11,25,30	17	34,50	14
10	8	8	12	—

Table 1.1 Chapters in some commonly used neuroscience texts that relate to chapters in Part I of *The Book of GENESIS* (BoG). The texts listed are *Neurobiology* (Shepherd 1994), *From Neuron to Brain* (Nicholls et al. 1992), *Principles of Neural Science* (Kandel et al. 1991) and *Foundations of Cellular Neurophysiology* (Johnston and Wu 1995).

Part II of the book is intended as an introduction to the use of GENESIS as a modeling system. After an introduction to our “modeling philosophy,” we begin a series of chapters that teach the use of the GENESIS simulation language for the construction of your own simulations. These generally follow the same sequence of topics as in Part I. Each chapter allows you to modify and build upon the simulations created in the previous one, as you progress from simpler to more complex models. The modular nature of GENESIS simulations encourages this approach to the development of simulations. Thus, you will be able to modify the tutorial simulations from Part I and use them as a starting point for your own original simulations. As some of these are based upon recent research simulations, this can drastically reduce the time required to produce a sophisticated neural simulation.

Finally, there are the appendices and indices. Appendix A describes the procedure for obtaining and installing the GENESIS simulator and the tutorials, and Appendix B gives listings of the various GENESIS simulation scripts that are used in Part II. In addition to the usual subject index, we have provided alphabetized indices for the GENESIS script language commands and basic simulation components (GENESIS objects) that are used in Part II.

One more comment on the general tone of this book. We have intentionally designed the tutorials and the text to provide a heavy emphasis on the actual “hands-on” use of the simulations described. Thus, it is written as a manual or “work-book,” rather than as a textbook to be read passively. Our emphasis on hands-on demonstrations reflects our conviction that this form of learning is more valuable than a standard lecture or reading format (Alper 1994).

Chapter 2

Compartmental Modeling

JAMES M. BOWER and DAVID BEEMAN

Before beginning to explore the tutorials, it is important to understand something about the assumptions and the mathematical models that underlie these simulations. Thus, although the first section of the book describes what are essentially “point and click” tutorials, it is important not to use these tutorials blindly. Their effective use requires some understanding of the basics of neural modeling, as well as the concepts in neuroscience that are introduced along with the tutorials. Entire books have been written on this subject, so obviously we can only highlight the issues here. However, throughout the text we have referenced other sources of information. If you are seriously considering building models yourself, we would strongly recommend that you consult these references.

2.1 Modeling Neurons

Figure 2.1A shows an example neuron based on a drawing of a pyramidal cell by Ramón y Cajal that we would like to model, either as a single cell, or as a component in a network of interacting neurons. This figure shows the tree-like structure of the dendrites, which receive synaptic inputs from other neurons. Synaptically activated ion channels in the dendrites create postsynaptic potentials that, we assume here for simplicity, are passively propagated to the pyramid-shaped cell body (soma) where voltage-activated ion channels may create action potentials. In most cells, these channels are concentrated near the base of the soma in the region called the *axon hillock* near the axon. The long axon at the bottom of the figure propagates action potentials to terminal branches that form synapses with other neurons. In some cases (Chapter 7) neurons may have voltage-activated channels in their dendrites.

This not only complicates their electrical properties and thus their simulation, but also is responsible for the complex dynamics of these neurons.

2.1.1 Detailed Compartmental Models

When constructing detailed neuronal models that explicitly consider all of the potential complexities of a cell, the increasingly standard approach is to divide the neuron into a finite number of interconnected anatomical compartments. Figure 2.1B shows a simplified model in which the neuron is divided into several dendrite compartments, a soma, and an axon. Each compartment is then modeled with equations describing an equivalent electrical circuit (Rall 1959). With the appropriate differential equations for each compartment, we can model the behavior of each compartment as well as its interactions with neighboring compartments.

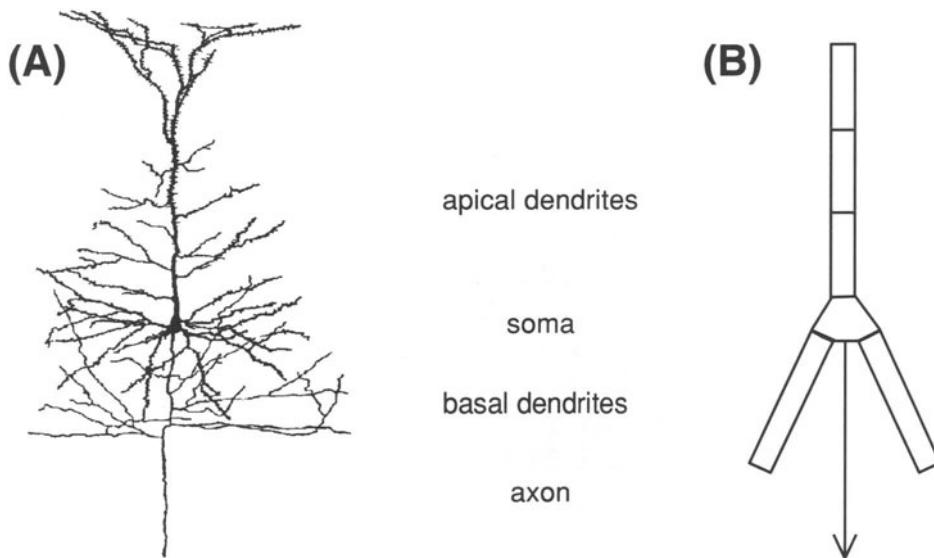


Figure 2.1 (A) A pyramidal cell with dendrites, soma, and axon. (B) A simplified discrete compartmental model of the same neuron.

In this type of detailed compartmental model, each compartment must be made small enough to be at approximately the same electrical potential. Often this means constructing simulations out of very large numbers of compartments. For example, we have recently published a GENESIS model of a cerebellar Purkinje cell that uses 4550 compartments and 8021 channels (De Schutter and Bower 1994a,b). The representation of this model is shown in Fig. 2.2.

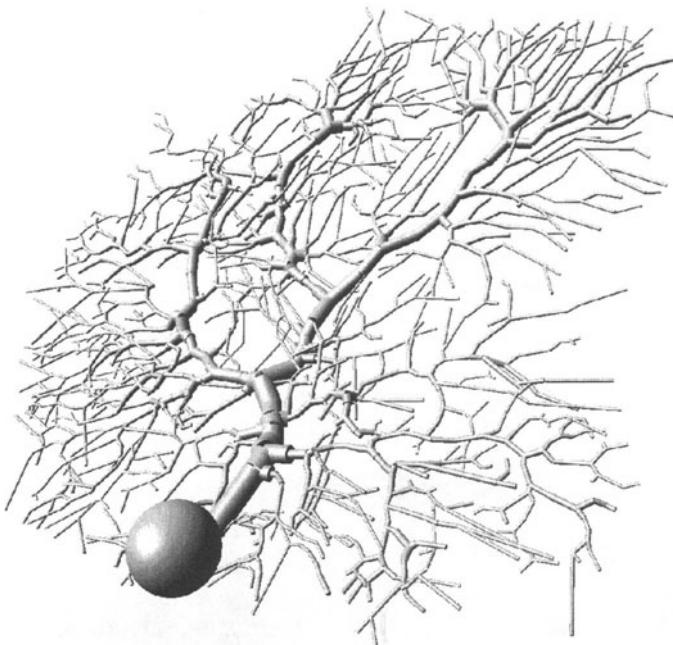


Figure 2.2 A detailed multi-compartmental model of a cerebellar Purkinje cell, created with GENESIS by De Schutter and Bower (1994a,b). Visualization by Jason Leigh using the GENESIS Visualizer program. The experimental data describing the cell morphology was provided by M. Rapp, I. Segev and Y. Yarom.

2.1.2 Equivalent Cylinder Models

For some purposes, it may be adequate to model neurons with a smaller number of non-equipotential compartments. Models of this sort can be used to model basic electrical properties of cells, or to construct small networks of neurons. Under these conditions, there are defined methods for constructing neuronal models dependent on the anatomy and physiology of the neuron in question. Many of these have been pioneered by Wilfrid Rall (cf., Segev, Rinzel and Shepherd 1995). For example, Rall has shown analytically that if dendritic trees approximately satisfy a “3/2 power law” and do not contain active conductances, they can safely be mapped into an equivalent linear structure (Chapter 5). In this way, under defined conditions, a complicated branching structure can be approximated by a much simpler linear dendrite model, as was done in the model shown in Fig. 2.1B. However, in general, as the known complexity of the physiology or anatomy of the neuron increases, it is usually necessary to revert to full-blown compartmental models.

2.1.3 Single and Few Compartment Models

In cases where large numbers of neurons are being placed in network models, limited computer resources sometimes require that neurons be modeled with single compartments or a very small number of compartments. For example, a large-scale GENESIS simulation of the olfactory cortex uses a network of 4500 neurons containing simple model pyramidal cells similar to the one shown in Fig. 2.1B (Wilson and Bower 1989, 1992). As you will see for yourself in Chapter 9, even such simplified neurons can sometimes capture experimentally observed behavior. In Chapter 7, we will see that only a single compartment is needed to model the behavior of some invertebrate “pacemaker” neurons. On the other hand, when building models of this type, one must always be aware that there are many local “computations” that occur in the extensive dendritic system of many neurons. Again, if these are of interest to the modeler, it is usually necessary to use hundreds or thousands of compartments.

2.2 Equivalent Circuit of a Single Compartment

Having described the general approaches to modeling single neurons, we now discuss in a bit more detail the basis for compartmental modeling. The reader should note that this section is intended as a very basic overview of neural modeling. The topics covered here are treated in more detail in Chapters 4–6 and by Segev, Fleshman and Burke (1989).

As we have described, the notion of an equivalent electrical circuit for a small piece of cellular membrane is the basis for all compartmental modeling. This arises from the fact that neuronal membranes have been demonstrated to behave as simple electrical circuits with some capacitance, resistance, and voltage sources. These model parameters define the so-called *passive properties* that are responsible for the way that electrical impulses are transmitted along the dendritic tree. It is generally necessary, as well as advisable, to begin all single cell modeling efforts with a consideration of passive cellular properties of the cell. These properties form the basis for the usually more interesting neuronal behavior that arises from the *active properties* provided by different voltage- or ligand-dependent conductances. If the passive properties are not modeled correctly, spurious results with active conductances are likely to be obtained.

Figure 2.3 shows the equivalent electrical circuit of a basic neural compartment. Here, V_m represents the membrane potential, or the potential in the interior of a compartment relative to a point outside the cell. The “ground” symbol at the bottom of the figure represents this external point, taken to be at zero potential. As the conducting ionic solutions inside and outside of the cell are separated by the cell membrane, the compartment acts as a capacitor. This is charged or discharged by current flowing into or out of the compartment. This current flow may be from adjacent compartments, from the passage of ions through channels in the cell membrane, or from current injection from an electrode inserted into the cell. The

membrane potential appears across the membrane capacitance C_m , and can cause a current flow into or out of the compartment at the left through the axial resistance R_a when there is a difference in potential $V_m - V''_m$ between the two compartments. Likewise, there may be a flow of current into or out of the primed compartment at the right through its axial resistance R'_a .

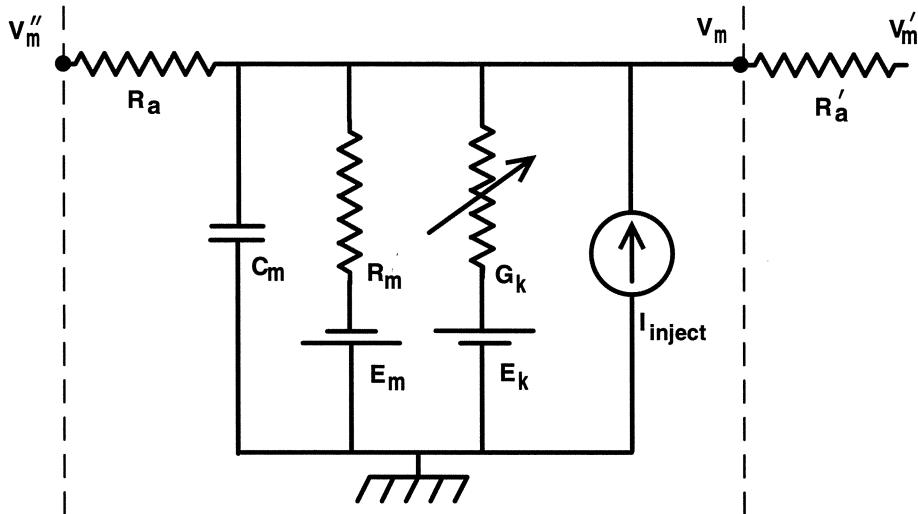


Figure 2.3 The equivalent circuit for a “generic” neural compartment.

The resistor with the arrow through it represents one of many possible variable channel conductances that are specific to a particular ion or combination of ions that give individual neurons and neuron types their unique computational properties. By convention, these are described in terms of the conductance G_k rather than the resistance. As the conductance is the reciprocal of resistance, the units of G_k are in reciprocal ohms, or siemens. Differences in the concentration of the ion between the inside and the outside of the cell result in an osmotic pressure which tends to move ions along the concentration gradient. The resulting charge displacement creates a potential difference that opposes this flow. The membrane potential at which there is no net flux of the ion is the *equilibrium potential* (or *reversal potential*) E_k , represented by a battery in series with the conductance. In the absence of synaptic input, current injection, or spontaneous firing of action potentials, V_m will approach a steady state *rest potential* E_{rest} , typically in the range of -40 to -100 mV. This is determined by the condition that there is no net current flow into the cell from the various types of ion channels.

The other resistor and battery linking the exterior and the interior of the cell represent the combined effect of passive channels (mainly those for chloride ions) having a relatively fixed conductance. The resistance is usually called the *membrane resistance* R_m , although it is

sometimes referred to as a *leakage conductance* $G_{\text{leak}} = 1/R_m$. The associated equilibrium potential E_m is typically close to the rest potential. In some cases, it is given a slightly different value, E_{leak} , in order to reduce the net channel current to zero when $V_m = E_{\text{rest}}$. Finally, the current source I_{inject} represents an optional injection current which could be provided by an electrode inserted into the compartment.

One may then calculate V_m using a differential equation which expresses the fact that the rate of change of the potential across C_m is proportional to the net current flowing into the compartment to charge the capacitance. On the right-hand side of Eq. 2.1, Ohm's law is used to calculate the current due to each of the sources shown in Fig. 2.3:

$$C_m \frac{dV_m}{dt} = \frac{(E_m - V_m)}{R_m} + \sum_k [(E_k - V_m)G_k] + \frac{(V'_m - V_m)}{R'_a} + \frac{(V''_m - V_m)}{R_a} + I_{\text{inject}}. \quad (2.1)$$

Here, the sum over k represents a sum over the different types of ion channels that are present in the compartment. The sign convention used in the GENESIS simulator defines a positive channel current to be one that causes a flow of positive charge *into* the compartment. The variable conductance of each channel type G_k gives the net effect of many individual channels that open and close in a binary manner.

To model this on a computer, we need to numerically solve Eq. 2.1 for each compartment. Of course, the V''_m and V'_m in the adjacent compartments affect the currents flowing into or out of the compartments, so we are solving many coupled equations in parallel. Also, we will need good models for the way that the conductances vary with voltage, time or synaptic input.

2.3 Axonal Connections, Synapses and Networks

Typically, but not always, neurons communicate by means of chemical synapses. The most common situation is one in which an action potential causes the release of a neurotransmitter from a presynaptic terminal at the end of an axon branch. This diffuses across a narrow gap to the postsynaptic junction (usually on a dendrite), causing an increase in conductance for a specific set of ion channels that are sensitive to this transmitter. However, synaptic connections may also be found between two axons or between two dendrites. In many cases, axons make connections to the cell body, rather than to dendritic branches.

Usually, we can treat an axon as a simple delay line for the propagation of action potentials, although it could also be modeled as a series of compartments if we were interested in understanding the details of axonal propagation. In most cases, the change in the postsynaptic channel conductance is a simple function of time which may be determined from experimental measurements. Then, we may avoid having to model the details of the process of presynaptic transmitter release, its binding to postsynaptic receptors, and the way

in which the permeability of the postsynaptic membrane is affected. Instead, we may use an analytical expression such as the *alpha function*, described in Chapter 6, to represent the resulting conductance of synaptically activated channels. When more detailed models of the biochemical reactions underlying synaptic transmission are required, they may be created using the techniques described in Chapter 10 and the graphical interface *Kinetikit*, which was created for the development of models of biochemical signaling pathways.

Of course, each synapse has associated with it an effect of a particular magnitude or “weight” on the postsynaptic cell. Furthermore, these weights can change under some circumstances. The implementation of this *synaptic plasticity* is discussed in Chapter 15. Chapter 19 treats one class of receptors, the voltage-dependent NMDA receptors, which have been shown to confer such weight-changing properties on synapses. Chapter 19 also considers electrical synapses, which are yet another type of connection between cells. Once we have modeled single neurons and the ways in which they may interact, we can proceed to modeling neural circuits and networks. Example models are discussed in Chapters 8 and 9, and the details of constructing your own network simulations are given in Chapter 18.

2.4 Simulation Accuracy

Once a modeler has constructed a simulation, the accuracy of the results depends on many factors, from the quality of the data used to construct the simulation, to the way in which the simulation is run numerically. In the case of an analytic solution, one knows the result is correct, assuming that the underlying model is correct. For numerical simulations, it is trickier. Is a surprising result the result of some error, or is it an exciting new discovery? How do we know when to trust a simulation? Throughout this book, we offer some suggestions for developing the sort of intuition and feeling for neuronal behavior that will help you to identify “suspicious” results and their possible causes. As described below, the causes can range from mistakes in the use of the simulation language (programming errors), conceptual errors in the model, inappropriate choices of parameters, and numerical inaccuracies due to the wrong size numerical integration step, to the legitimate but unanticipated behavior of a complex system.

2.4.1 Choice of Numerical Integration Technique

A neural simulation program solves a set of coupled equations like Eq. 2.1 by replacing the differential equation with a difference equation that is solved at discrete time intervals. Typically, smaller time intervals lead to greater accuracy but slower execution time, as more time steps are required for the solution over a given time period. A wide variety of numerical integration techniques has been developed to carry out this procedure with the best compromise between speed and accuracy. These fall into two general categories. So-called

explicit methods are the simplest, but can require very small time steps in order to avoid numerical instabilities when there are many small compartments in a model. The *implicit* methods are more complex, but are much more stable (Mascagni 1989).

GENESIS provides a choice between several different numerical integration methods, ranging from the crude explicit *forward Euler* method through highly stable implicit methods. These methods are described in Chapter 20. In general, the best method to use depends on the nature of the model. Often, there is a tradeoff between ease of use and computational efficiency. The default integration method is the *exponential Euler* method, which is optimized for the solution of equations that are of the general form of Eq. 2.1 (MacGregor 1987). This is generally the best choice for cell models having only a few compartments, as used in most network simulations. Chapter 20 discusses the use of a generalized version of the algorithm developed by Hines (1984) for implementing the *backward Euler* and *Crank-Nicholson* implicit methods. These are the fastest of the numerical methods used by GENESIS and are stable and accurate when used with relatively large integration steps. These are used for detailed cell models that contain many compartments. However, they require some additional steps in setting up the simulation and make it harder to interactively modify the simulation. For this reason, one usually develops and refines a simulation using the default method and switches to one of the implicit methods if additional speed is needed for long simulation runs. The *Cable* tutorial which is described in Chapter 5 allows you to experiment with the various integration methods that are available.

2.4.2 Integration Time Step

Even after you have selected a numerical integrator, the step size used in the numerical integration of the differential equations that describe the model is important. The difficulty is that certain combinations of parameters can sometimes result in a situation where the step size is too large to yield accurate results. On the other hand, the use of too small a time step may lead to round off errors, as well as unnecessarily slow execution of the simulation. In general, the step size should be much smaller than the time scale for the most rapidly occurring events. For example, the action potentials that are produced in a simulation typically rise to their maximum value in about 1 msec. Thus, a time step of 0.01 msec is an appropriate choice.

The default value of the time step that is given for these tutorials usually results in a good compromise between accuracy and speed of computation. However, if you have made significant changes in the default parameters for a simulation, it would be a good idea to experiment with the step size. If increasing it makes no changes in the results, you can use the larger step size to speed up the simulation. If decreasing the step size causes the results to change, you should continue to decrease it until you see no significant changes. Several exercises in the tutorials that follow deal with this question.

2.4.3 Accuracy of GENESIS

Finally, even if a modeler has been extremely careful in constructing a particular model, its accuracy is still dependent on the software in which it is coded. This is especially the case if one is using a general-purpose simulation system such as GENESIS. For this reason, when choosing a neural simulator, the reliability and accuracy of the simulator is at least as important a consideration as its speed. To quantify the speed and accuracy of both GENESIS and other simulators, we have developed the “Rallpacks” suite of benchmarks (Bhalla, Bilitch and Bower 1992). The Rallpacks are currently based on three sets of benchmarks: a linear passive cable with many compartments; a highly branched cable; and a linear axon containing Hodgkin–Huxley channels. In the first two cases, simulator results can be compared to exact analytic solutions. In the third case, which already is an example where the complexity of the model makes analytic solutions impossible, results can be compared to other frequently used, but independently developed simulators. These measures demonstrate that GENESIS is as fast and accurate as any existing simulation system.¹ However, as models within GENESIS or any simulation system become more complex, modelers must be more and more skeptical, vigilant and self-critical.

¹The set of Rallpack benchmarks may be obtained by *ftp* in the same manner as GENESIS, following the procedure outlined in Appendix A.

Chapter 3

Neural Modeling with GENESIS

JAMES M. BOWER and DAVID BEEMAN

3.1 What Is GENESIS?

Now that we have briefly described the numerical basis for the tutorials included in the first part of this book, we are ready to get started with running the tutorials. The tutorials included in this manual are all constructed using GENESIS, the General NEural SImulation System that has been under development in our laboratory at Caltech since 1985. This chapter is intended to introduce each of the tutorials, as well as provide a demonstration as to how to use the GENESIS graphical interface. First, however, we provide some basic information about the GENESIS system on which the tutorials are based.

3.1.1 Why Use a General Simulator?

In principle, each of these tutorials and their graphical interfaces could have been written as a dedicated piece of software not dependent on a general simulation system (cf., Huguenard and McCormick 1994). In this case, each tutorial would have included only the code necessary to run the particular simulation. If the simulation were written well, this would result in the maximum use of the speed and memory of the computer being used. However, as the speed of computers has increased and computer memory has become cheaper and cheaper, the advantage of dedicated simulation code as compared to general simulators has become less and less clear. In addition, a well-designed neural simulator can provide very good performance, even when compared to dedicated code. We have estimated, for

example, that the overhead costs associated with GENESIS average to less than a factor of two in simulation speed.

With a usually small compromise in simulation speed, simulation systems like GENESIS can actually provide many important advantages over dedicated code. Many of these advantages result in a dramatic speedup in the process of building and expanding models that usually more than compensates for the slight reduction in simulation speed. Almost always, the time taken to build a model is considerably greater than the time taken to actually simulate it, once built. Furthermore, adding new components to the simulation does not require rewriting all the existing code, as is often the case with dedicated simulations.

A second advantage of general-purpose simulators is that model components can be shared between different simulations. The ability to use previously developed and tested components leads to tremendous speedups in the time necessary to develop new simulations. The second part of this book demonstrates the ease with which GENESIS allows such modifications.

Beyond issues of flexibility and speed of model construction, there are also several technical advantages to general simulation systems. These include some assurance that the software core of the simulations themselves are accurate, more accessibility to new simulation-related technology such as parallel computers (Nelson, Furmanski and Bower 1989), new integration techniques (Hines 1984), or new forms of graphics (Leigh, De Schutter, Lee, Bhalla, Bower and DeFanti 1993), and some form of standardization for model descriptions.

Perhaps most importantly, general simulation systems such as GENESIS can lead to fundamentally new mechanisms of communication between those interested in how the brain computes. For example, by using GENESIS, it is possible to test the modeling results of others by actually running their simulations. The GENESIS users' group, BABEL, maintains a database of published simulations for this purpose. Several of the tutorials in this manual were modified from research simulations. In addition, simulations can serve as a means of communication between different laboratories interested in the same system. We have recently pointed out (Bower 1992) that simulation systems and their component libraries can become a form of neural database that not only represents structural information about the nervous system, but does so in such a way that structural details are functionally organized. Thus, when a general simulation system is properly designed, the more it is used, the more information about the nervous system is included. Finally, a system such as GENESIS allows books like this one to be written, leading, we hope, to a more efficient and interesting way to learn about the organization of the nervous system.

3.1.2 GENESIS Design Features

GENESIS was developed primarily as a means of constructing biologically realistic neuronal simulations. In addition to our interest in supporting the general advantages of simulators just

described, three other basic objectives determined our design philosophy: (1) the simulator must be capable of addressing problems at many levels of detail (i.e., parts of neurons to large neural systems); (2) the system should be open-ended, placing few limits on the kinds of problems that can be addressed; and, (3) the system should be user-extensible to allow the incorporation of new modeling efforts. The following specific features of the system have evolved, based on the following criteria.

Modular Design

As a key principle conferring both flexibility and organization on GENESIS, we adopted a highly object-oriented programming approach. Simulations are constructed of “building blocks” or modules, each of which perform well-defined functions and have standardized means for communicating with each other. The level of detail of a simulation is then determined by the nature of the building blocks one chooses. The user has complete freedom to assemble a model with any set of modules. Object-oriented design also enables the user to easily add new modules to extend GENESIS for a particular application.

Flexible Interface

The GENESIS user interface consists of two parts. The underlying level is the Script Language Interpreter, or SLI. This is an interpretive programming language similar to the UNIX shell, with an extensive set of commands related to building simulations. The interpreter can read SLI programs (scripts) either interactively from the keyboard, or from files. The graphical interface is XODUS, the X-windows Output and Display Utility for Simulations. This provides a higher level and user-friendly means for developing simulations and monitoring their execution. XODUS consists of a set of graphical modules that are exactly the same as the computational modules from the user’s point of view, except that they perform graphical functions. As with the computational modules, XODUS modules can be set up in any manner that the user chooses to display or enter data. Furthermore, the graphical modules can call functions from the script language, so the full power of the SLI is available through the graphical interface.

Device Independence

In order to make the simulator portable and available to the maximum number of users, the system has been designed to run under UNIX and X-windows and has been compiled and tested on a number of machine architectures. This facilitates communication by enabling groups utilizing different machines to run each other’s simulations. A parallelized implementation has also been developed that dramatically extends the range of problems which GENESIS can address. The use of parallel GENESIS is described in Chapter 21.

3.1.3 GENESIS Development

GENESIS was initially developed in our laboratory by Matthew Wilson as an extension of efforts to model the olfactory cortex (Wilson and Bower 1989, 1992). The GENESIS graphical interface, XODUS was subsequently developed primarily by Upinder Bhalla as a general-purpose user interface for the GENESIS system. Since that time, many additional modelers both at Caltech and at other institutions have contributed to its design.

3.2 Introduction to the Tutorials

One of the interesting features of neural modeling is that it is possible to start at almost any scale or level of detail. In the case of realistic modeling, it is usually the case that whatever level one begins with, the results themselves push the modeler to consider other levels. Thus, detailed single cell modeling quickly requires a better understanding of the network in which the cell is embedded. On the other hand, network level modeling often serves to focus attention on the details of single cell physiology.

As we have stated, GENESIS was specifically designed to allow and promote this multi-scale modeling within a single simulation system. At present, it is the only simulator with this capability. The tutorials described in Chapters 4–10 each represent simulations at different levels. The sequence also reflects the way in which large-scale simulations can be built up from more detailed simulations. All of these simulations can be performed without any knowledge of the GENESIS simulation language. The second half of this book retraces this sequence of topics with a series of tutorials and exercises that are designed to teach you how to program your own simulations with GENESIS, emphasizing the process whereby any GENESIS model can be expanded and changed into a new simulation. The tutorials presented in the first part of this book, and introduced briefly below, will first familiarize you with the various modeling levels supported in GENESIS and used by computational neurobiologists to understand neuronal organization.

Chapter 4 describes the experiments and mathematical models used by Hodgkin and Huxley to understand the time- and voltage-dependence of the ionic conductances responsible for the generation of action potentials. In this case, the tutorial simulation represents a single piece of a neuronal axon. However, these Hodgkin–Huxley forms of voltage activated channels are an important ingredient of many more complex neuronal models. Most model tuning involves manipulation of the Hodgkin–Huxley parameters governing ionic channels. The associated equations also usually represent the most computationally intensive components of single cell simulations. In this tutorial, a GENESIS re-creation of the Hodgkin–Huxley model is used to perform current and voltage clamp experiments and to understand the basis of postinhibitory rebound and neuronal refractory periods.

Chapter 5 explores the “cable properties” of a series of passive compartments without active ion channels. In this case the tutorial has linked together several compartments of

the type presented in Chapter 4, but without the Hodgkin–Huxley channels. Although such models do not include active channels, which are one of the principal features of the nervous system, there is still a great deal that may be learned from such models. Studies of the passive properties of neurons lay the foundation for much more complex simulations. They can also be used with experimental measurements to determine the morphology of a cell.

Chapter 6 combines elements of Chapters 4 and 5 into a multi-compartmental neuronal model. This tutorial also introduces models of synaptically activated channels. The theory presented, along with the simulation *Neuron*, explores the effects of temporal summation of multiple synaptic inputs. The neuron contains spatially separated dendrite compartments with both excitatory and inhibitory synaptically activated channels, and a soma with Hodgkin–Huxley sodium and potassium channels.

Chapter 7 uses two different simulations to understand how interactions between several different types of voltage-activated channels can lead to more complex firing patterns in single neurons. In one simulation, a model of the soma of a molluscan pacemaker cell, a particular combination of somatic channels generates periodic bursts of action potentials. In a second model of a hippocampal pyramidal cell, bursting action potentials are shown to arise through an interaction between the soma and more distant dendritic regions.

Chapter 8 introduces for the first time networks of neurons. In this case, we explore a simple network taken to represent a central pattern generator, or CPG. Circuits of this sort have been shown to control many types of rhythmic behavior in a wide range of different animal species. As is often the case with multi-neuron network models, the CPG model considered here is more abstract than an actual biological network that might produce similar firing patterns. In this tutorial, we use GENESIS to explore the circuit interactions between four neurons that are similar to the model neuron used in Chapter 6. The behavior of this network is compared to mathematical models of simple coupled oscillators. The network may also be used to replicate the various footfall patterns of a horse.

Chapter 9 incorporates many of the features explored in the previous chapters into a more complex multi-neuronal network model. In this case, the model is intended to allow you to explore several aspects of the dynamical properties of cerebral cortical networks. Unlike the idealized network used in Chapter 8, this simulation attempts to both realistically represent basic features of real neurons, while at the same time linking them together into a sizable network. The result is a fairly complex simulation that also takes longer to execute than any of the other tutorials. The simulation itself is a “user-friendly” version of the research simulation used by Wilson and Bower (1989, 1992) to model the neuronal dynamics of the mammalian olfactory cortex.

We conclude Part I of this book with Chapter 10, returning to the subcellular level of modeling to consider a different type of network — the interacting biochemical signaling pathways that underlie the processes of synaptic transmission and channel activation. The theory of biochemical computation and the use of the *Kinetikit* graphical interface are discussed in this final chapter.

3.3 Introduction to the GENESIS Graphical Interface

Before we explore the first tutorial, it is first necessary for you to understand the basics of the GENESIS graphical interface, XODUS. By using this interface, it is possible to easily investigate a wide variety of models and to vary their parameters without doing any programming. XODUS allows many different ways of visualizing the state of the many relevant variables that are present in neurobiological simulations. In order to give you a feel for the use of XODUS, we use the tutorial *Neuron*, which is described in Chapter 6. This tutorial is intended for use in understanding the effects of synaptic inputs to a model neuron. However, its user interface is fairly representative of most of the tutorials used in this book.

3.3.1 Starting the Simulation

In this book, it is assumed that GENESIS and its associated scripts for the tutorials have been installed on a workstation that uses the UNIX operating system and some variety of X Windows. Appendix A describes the procedure for acquiring and installing the software if you have not already done so. We recommend that if you are not fully familiar with UNIX and its file system, that installation of GENESIS be carried out by the system administrator or person responsible for maintaining your machine. Typically, the tutorials and other GENESIS scripts will be installed in subdirectories of the */usr/genesis/Scripts* directory. We will refer to this simply as the *Scripts* directory.

Although you will get a chance to explore this tutorial more thoroughly in Chapter 6, it would be a good idea to run the simulation now, in order to become familiar with the procedures that we will use in the following chapters. In order to use GENESIS, you will obviously have to first login to a UNIX workstation with access to the GENESIS scripts. Follow your local procedures for logging in and creating a “terminal window” on your workstation. So that it will not be completely covered by the simulation display, move the window to the lower left corner of the screen. This is usually accomplished by holding down the left mouse button while the cursor is on the title bar and dragging the mouse to move the window.

The next step in running a GENESIS simulation involves changing your directory to that containing the GENESIS tutorial of interest. In keeping with the modular structure of GENESIS simulations, most tutorials consist of a short main simulation script that invokes several other scripts. This collection of scripts is kept in its own subdirectory of the *Scripts* directory. Although this practice is by no means universal, it is customary to give the main simulation script a name that starts with a capital letter in order to distinguish it from the other associated scripts. The script that runs the *Neuron* simulation used here for illustration is called *Neuron.g*. It is kept in the *Scripts/neuron* directory and is invoked by typing “*genesis Neuron*” after changing to this directory. The command for starting

other GENESIS tutorials is of the same form. Thus, instructions for starting tutorials in this book typically start out by suggesting something like “change to the *Scripts/neuron* directory”. This should be interpreted to mean that you should give the UNIX command “`cd /usr/genesis/Scripts/neuron`”. If *Scripts* has been placed elsewhere, you may need to specify a different path.

Once you are in the correct directory, type the command “`genesis Neuron`”. (Don’t forget that UNIX is case-sensitive, so you will have to type the capital N!) If the files have been properly installed and the necessary paths have been set, you should be rewarded with some messages which indicate that GENESIS is loading. Eventually, a number of graphs and a “control panel” will appear on the screen. (If not, you or your system administrator should consult Appendix A, “Acquiring and Installing GENESIS”.)

3.3.2 The Control Panel

Figure 3.1 shows the left-hand portion of the control panel for the tutorial *Neuron*, which should now be present near the bottom of your screen.

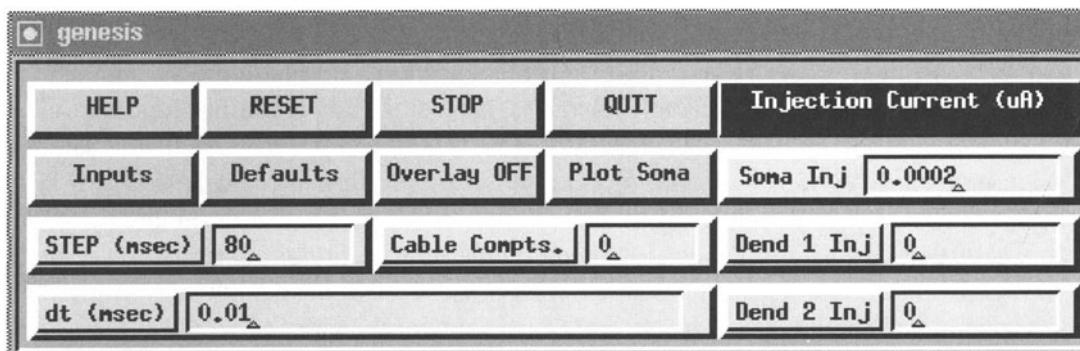


Figure 3.1 Part of the *Neuron* control panel, showing the various types of XODUS widgets that are used to control the simulation.

The control panel and the other windows that appear are examples of GENESIS *forms*. Like the terminal window and the other windows that are used with X Windows, they may be moved and resized by clicking and dragging with the mouse. This is particularly useful if the resolution of your display doesn’t allow everything to be visible on your screen. Try moving the control panel up and down and resizing it. (The procedure will vary depending on the window manager that is used with your system, but forms are normally resized by clicking on the icon at the right end of the title bar and dragging the mouse.) The several rectangular areas in the control panel are XODUS *button*, *label*, *dialog box* and *toggle* widgets. On a color display, these four types of widgets will each have their own distinctive colors. Most simulations have buttons corresponding to the HELP, RESET, STOP and QUIT buttons shown

here. Clicking the left mouse button on one of these will cause GENESIS to perform the specified action. (Unless otherwise stated, the instructions “click on ...” will mean “click the left mouse button on”)

The three dialog boxes underneath the **Injection Current** label are used to display information or to enter data from the keyboard. For example, the **Soma Inj** dialog box is used to hold the value of the injection current that is being applied to the soma in this model (see Chapter 6). To change the data field of any particular dialog box, use the mouse to move the cursor into the dialog box. Then use the keyboard right and left arrow keys to move the “ \sim ” symbol to the right of the place where you wish to make the change. Then use the “Delete” key to back up over anything you wish to change, and type in the changes. To cause the changes to be entered and acted upon, you must then hit “Return” (or “Enter,” on some keyboards) while the cursor is in the data field. (NOTE: forgetting to hit “Return” after changing the contents of a dialog box is a common mistake for novice GENESIS users. If you ever suspect that the old values are being used in the running of a simulation, move the cursor back into the dialog box, hit “Return,” and repeat the run.) Clicking the mouse on the label field at the left of a dialog box has the same effect as hitting “Return” in the data field. This is often done when you don’t want to change the data in the dialog box, but you would like the simulation to act upon the existing data. For example, the **STEP** dialog is used to both enter the number of milliseconds of simulation time and to run the simulation for this amount of time. Try clicking on the **STEP** label of the box or hitting “Return” when the cursor is in the data field containing “80.” This should perform the default simulation, which is to apply a 0.2 nA current injection pulse to the soma of the cell. When it has finished, click on **RESET** to reset the simulation to time step 0 and clear the graphs.

The dialog box labeled **dt** gives the step size to be used in the numerical integration of the differential equations that describe the model. Although the default value is usually appropriate, you should pay attention to the considerations mentioned in Sec. 2.4.2 when making significant changes in the simulation parameters.

The rectangular areas labeled **Overlay OFF** and **Plot Soma** are examples of toggle widgets. Clicking on a toggle causes the label and the associated state of the toggle to alternate between two values. For example, after changing to **Overlay ON**, the graphs will not be cleared after clicking on **RESET**. This is useful when you wish to compare the results of two runs using different parameters. Click on **Plot Soma**. What happens?

3.3.3 Using Help Menus

In many cases, clicking on a button causes another window or menu with its own widgets to appear. This is the case with the **HELP** button. Each tutorial has a **HELP** button that may be used to call up a menu of topics. Both scrollable text windows and image windows are available. The latter are used for drawings that illustrate the experimental arrangement of the model being used, or to display typical experimental results scanned from journal articles.

Figure 3.2 shows the help menu for this tutorial at the right.

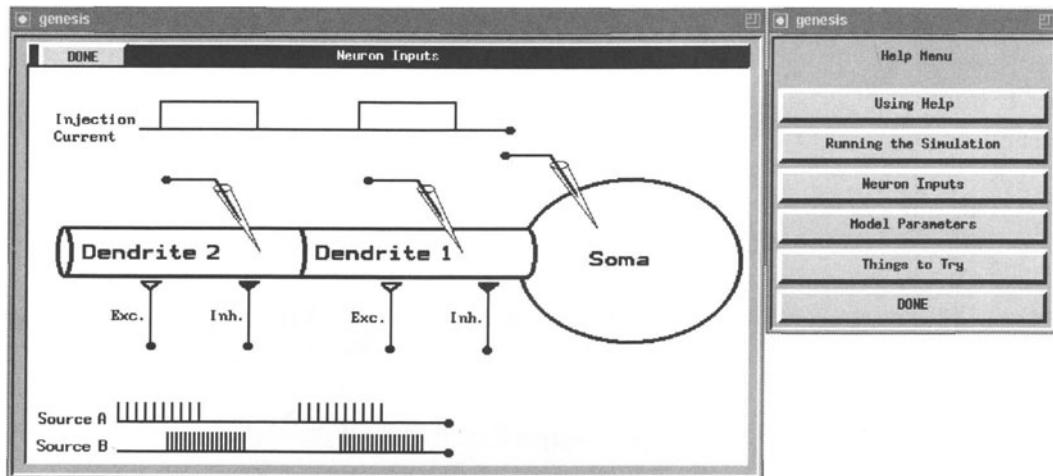


Figure 3.2 The HELP menu on the right has been used to call up the Neuron Inputs diagram on the left.

The drawing at the left was brought up by clicking on the Neuron Inputs button in the help menu. Many of the tutorials have a help menu selection similar to Running the Simulation. This brings up a text window with a description of the model that is depicted in the Neuron Inputs diagram, as well as a description of each of the control panel widgets.

In this particular tutorial, we have a soma and two dendrite compartments that are connected by axial resistances, as shown in Fig. 2.3 (page 11). The soma has voltage-activated channels like the ones modeled by Hodgkin and Huxley and described in Chapter 4. The dendrite compartments have both excitatory and inhibitory synaptically activated channels that respond to spikes applied at the synapses. These spike trains represent possible inputs delivered from the axons of other neurons. Chapter 6 uses this simulation to illustrate the properties of these types of channels. The diagram illustrates the different types of stimulation that we can apply. For example, we can inject pulses of current into any of the compartments, or connect spike trains to any of the synapses with a specified synaptic weighting. The Inputs button in the control panel brings up a window with dialog boxes to set the timing of the injection pulses and the bursts of spikes from sources A and B.

In any particular model or tutorial, there are typically many parameters of the model that one would like to vary. Other buttons in the control panel (not shown in Fig. 3.1) bring up menus with dialog boxes for changing the values of many of the parameters that characterize the variables in Eq. 2.1. Most of the tutorial simulations have help menu selections similar to Model Parameters that describe these options. Usually, there is a selection like Using Help that describes the use of the help system and how to move through the text in the text

windows. These secondary “pop-up” menus will have a button (usually labeled DONE, or Dismiss) that is used to hide the window again. When you have finished with the Neuron Inputs diagram, click on the DONE button. Likewise, clicking on the DONE button in the help menu hides it from view.

In some tutorials, including *Neuron*, the interface has been designed to allow the model itself to be altered. In *Neuron*, for example, one can put any number of passive “cable” compartments between the two dendrite compartments. These are similar to the dendrite compartments, but have no variable channel conductances. This will let us see what happens if we have spatially separated inputs to the neuron. Chapter 5 treats the passive propagation of electrical signals in neuronal cables in considerable detail. In the *Neuron* tutorial, cable compartments are added by entering a number in the Cable Compts dialog box. Other menus (described in Sec. 6.5) allow you to vary the parameters of these compartments.

3.3.4 Displaying the Simulation Results

GENESIS can use the XODUS graph widgets to plot any quantity of interest. For example, in *Neuron* one can look at the membrane potential or channel conductances of any particular compartment. Figure 3.3 shows the results of performing one of the experiments described in the Things to Try help menu selection. This experiment is also described in Chapter 6. A train of spikes at 2 msec intervals is delivered to the excitatory synapse in the first dendrite compartment and a burst of spikes to the inhibitory synapse arrives 10 msec later. The plots show the channel conductances and membrane potential in this compartment and in the soma.

XODUS allows flexible control of all the graphs it generates. For example, the scales of the graphs can be changed by clicking on the scale button, changing the values in the appropriate dialog boxes which appear, and pressing the DONE button. For example, you may wish to increase the time scale (xmax) if you click on STEP more than once without resetting, or you may wish to view a more limited range of data in order to increase the resolution. The existing data will be replotted whenever the scale is changed.

In addition to generating plots like those shown in Fig. 3.3 or presenting images like the diagram in Fig. 3.2, XODUS also has the ability to generate a representation of a cell or a network of cells that can be used to interact with the simulation or to view some quantity of interest throughout the cell or the network. In several of the simulations, color is used to display the propagation of action potentials throughout a multi-compartmental model and to identify the compartments in which they are generated. An example of this can be found in Chapter 7.

Figure 7.3 represents a model of a hippocampal CA3 pyramidal cell. The mouse was used to plant an injection electrode in the soma, and recording electrodes in the soma and a position in the apical dendrite. In this gray-scale rendering of the color display, the light region to the right represents areas in the apical dendrite where calcium channels generate

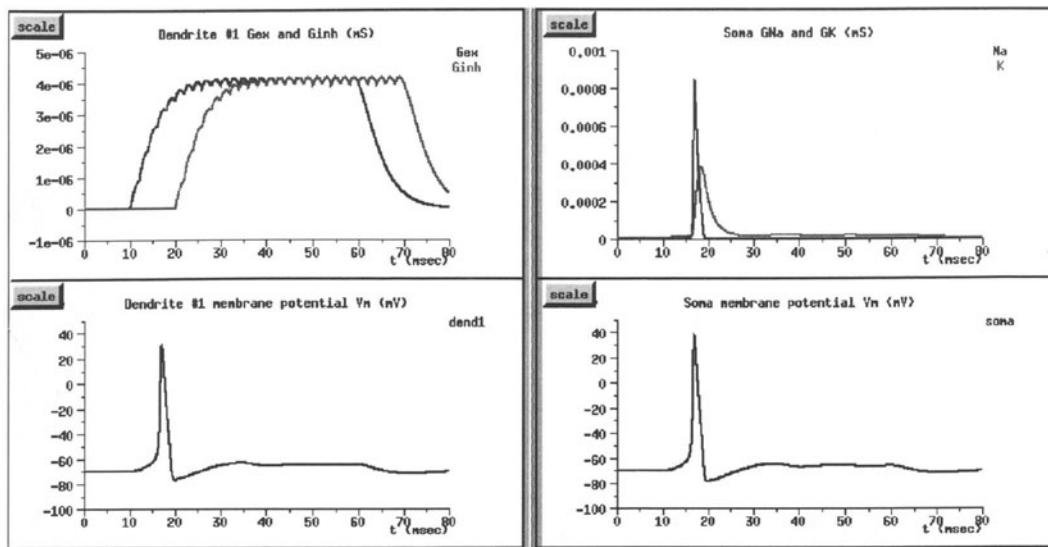


Figure 3.3 Some of the graphs that are produced after providing a combination of excitatory and inhibitory inputs to the model neuron.

action potentials. These are triggered by the propagation of action potentials from sodium channels in the soma. In Chapter 7, we discuss the interesting interactions between different voltage-activated channels that lead to the behavior shown in the figure. However, our first step is to understand how voltage-activated sodium and potassium channels can generate action potentials. This is the subject of the following chapter.

Chapter 4

The Hodgkin–Huxley Model

MARK NELSON and JOHN RINZEL

4.1 Introduction

Our present day understanding and methods of modeling neural excitability have been significantly influenced by the landmark work of Hodgkin and Huxley. In a series of five articles published in 1952 (Hodgkin, Huxley and Katz 1952, Hodgkin and Huxley 1952a–d) these investigators (together with Bernard Katz, who was a coauthor of the lead paper and a collaborator in several of the related studies) unveiled the key properties of the ionic conductances underlying the nerve action potential. For this outstanding achievement, Hodgkin and Huxley were awarded the 1963 Nobel Prize in Physiology and Medicine (shared with John Eccles, for his work on potentials and conductances at motoneuron synapses). The first four papers in the series summarize an experimental *tour de force* in which Hodgkin and Huxley brought to bear new experimental techniques for characterizing membrane properties. The final paper in the series places the experimental data into a comprehensive theoretical framework that forms the basis of our modern views of neural excitability. For a discussion and review of these seminal papers, see Rinzel (1990).

Hodgkin and Huxley indeed were aware that their findings and ideas had broad implications; they implicitly acknowledged this by the title of their fifth paper (Hodgkin and Huxley 1952d), which was the only one in the series not to explicitly mention the squid by name. Although the squid giant axon ultimately may have served as a means to an end, this is not to deny the squid her proper credit. Her generosity in providing a technically convenient preparation — a gargantuan axon, up to 1 mm in diameter — is often acknowledged. Less widely

appreciated is the fortuitous fact that, relative to most excitable nerve membrane, the squid axon is a simple system with basically only two types of voltage-dependent conductances. Today, we know of many more conductance types that can contribute to the excitability of nerve cells (Llinás 1988; also, see Chapter 7 in this volume). The squid axon membrane was an ideal model system; it presented a suitably generic and tractable problem, the solution of which gave rise to powerful new techniques and fundamental concepts.

In this chapter, we explore the Hodgkin–Huxley (HH) model using a GENESIS tutorial simulation called *Squid*. Before describing the mathematical model and performing the simulations, we provide a brief historical overview, so that the reader may better appreciate the scientific impact this work had at the time and how it has come to shape our present understanding of neural excitability. In exploring the HH model in this chapter, we will only be able to touch on some of the highlights. For a fuller appreciation of the model, we recommend a careful reading of the original paper (Hodgkin and Huxley 1952d). Additional historical and biophysical background on the HH model may also be found in Cole (1968), Hodgkin (1976), and Hille (1984).

4.2 Historical Background

For perspective, we begin by recounting the experimental evidence and theoretical concepts about neural excitability that existed at the time when Hodgkin and Huxley were developing their ideas and techniques. At that time, it was known that nerve cells had a low-resistance cytoplasm surrounded by a high-resistance membrane, that the membrane had an associated electrical capacitance, and that there was an electrical potential difference between the inside and the outside of the cell. It is important to note that until about 1940, there was no way to measure the membrane potential directly. Prior to that time, observations of nerve cell activity were made only with extracellular electrodes, which are capable of detecting electrical activity and action potentials, but only provide indirect information about the membrane potential itself.

A key piece of experimental data on neural excitability was obtained when Cole and Curtis (1939) used a Wheatstone bridge circuit to obtain the first convincing evidence for a transient increase in membrane conductance during an action potential. Their results were generally consistent with a popular hypothesis proposed much earlier by Bernstein (1902) that predicted a massive increase in membrane permeability during an action potential. Bernstein had formulated his hypothesis by reasoning as follows. It was known that a cell's membrane separated solutions of different ionic concentrations, with a much higher concentration of potassium inside than outside, and the opposite for sodium. By applying Nernst's theory, Bernstein was led to suggest that the resting membrane was semipermeable only to potassium, implying that at rest the membrane potential V_m should be close to the potassium equilibrium potential E_K of about -75 mV . Then, during activity, he believed

that a “breakdown” in the membrane’s resistance to all ionic fluxes would occur and the potential difference across the membrane should disappear; i.e., V_m would tend to zero. Although the conductance increase observed by Cole and Curtis (1939) during the action potential was qualitatively consistent with Bernstein’s hypothesis, the increase was not as large as one would expect from an extensive membrane breakdown.

During a postdoctoral visit in the U. S. spanning 1937–38, Hodgkin established ties with Cole’s group at Columbia University and worked with them also at Woods Hole in the summer. He and Curtis almost succeeded in measuring V_m directly by tunneling along the giant axon with a glass micropipette (Fig. 4.1A). Eventually, both Hodgkin and Curtis succeeded in this endeavor, albeit with other collaborators (Curtis and Cole 1940, Hodgkin and Huxley 1939), and they found not only did V_m rise transiently toward zero, but surprisingly there was a substantial overshoot, such that the membrane potential actually reversed in sign at the peak of the action potential (Fig. 4.1B). This result brought into serious question Bernstein’s simple idea of membrane breakdown and provided much food for thought during the span of World War II when Hodgkin, Huxley, and many other scientists were involved in the war effort.

Further insights into the nature of the membrane changes that occurred during an action potential required the development of two important experimental techniques referred to as the *space clamp* and the *voltage clamp*. The space clamp technique was developed by Marmont (1949) and Cole (1949) as a means of maintaining a uniform spatial distribution of membrane voltage V_m over the region of the cell where one was attempting to measure the membrane current. This could not be accomplished using the intracellular capillary electrode technique that had been developed to directly measure membrane potentials (Curtis and Cole 1940, Hodgkin and Huxley 1939). The tip of the capillary electrode acted essentially as a point source of current that would flow intracellularly along the axon, away from the recording site and not just through the membrane near the electrode. To achieve space clamping, the axon was threaded with a silver wire to provide a very low axial resistance, thereby eliminating longitudinal voltage gradients.

In conjunction with the space clamp, Cole and colleagues were also developing the voltage clamp technique that would allow the membrane potential to be maintained at any desired voltage level. One might think this simply would be a matter of connecting a constant voltage source across the cell membrane using a pair of electrodes, one inside and one outside the cell. In practice, this simple approach doesn’t work particularly well because of unpredictable voltage drops that occur in the solutions surrounding the electrodes. The technique that was eventually developed involved two pairs of electrodes. One pair was used to monitor the voltage across the membrane and the other was used to inject enough current to keep the measured voltage constant. In order to keep pace with the rapid changes in membrane permeability, the injected “clamping current” was controlled using a feedback amplifier circuit (Fig. 4.2A).

In order to take full advantage of the space clamp and voltage clamp techniques, it was

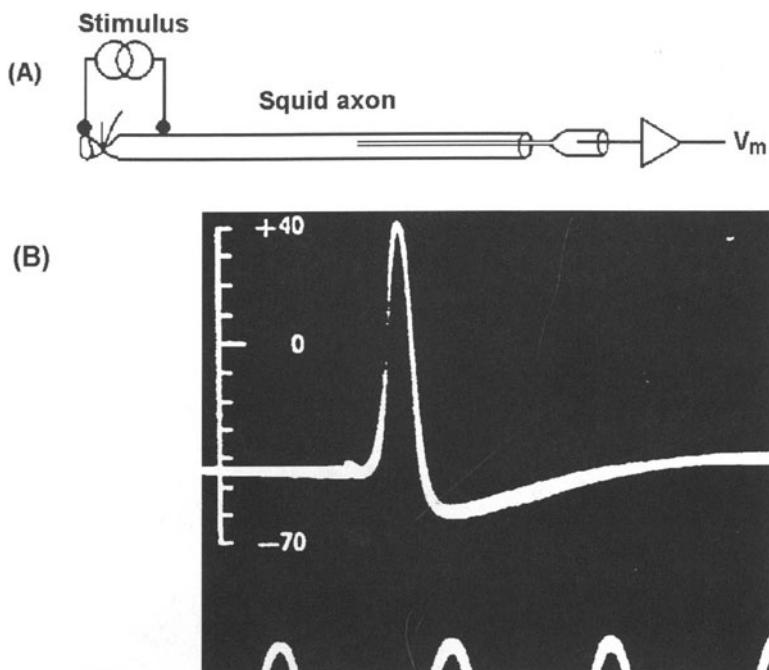


Figure 4.1 First direct measurements of membrane potential in squid giant axon. (A) Capillary tube filled with sea water has been carefully pushed down axon and serves as electrode to measure potential difference across membrane (after Hille 1984). (B) Membrane voltage V_m (in mV) during action potential. Time indicated by 500 Hz sine wave on oscilloscope screen. (Adapted from Hodgkin and Huxley (1939); reprinted with permission from *Nature*, Copyright 1939, Macmillan Magazines Limited.)

necessary to develop a means for identifying the individual contributions to I_{ion} from different ion species. Work by Hodgkin and Katz (1949) had demonstrated that both sodium and potassium made important contributions to the ionic current. This work also helped explain the earlier puzzling observations that V_m overshoots zero during the action potential. In contrast to Bernstein, who imagined the action potential to result from an unbounded transient increase in permeability for all ions, Hodgkin and Katz realized that bounded changes in permeabilities for different ions could account for the observed changes in V_m . In their view, V_m would tend to the Nernst potential for the ion to which the membrane was dominantly permeable, and this dominance could change with time. For a membrane at rest, they agreed with Bernstein, that the potassium conductance is overriding, and hence the resting potential is near E_K (about -75 mV). But during the action potential upstroke, they postulated that a dramatic shift took place, causing the membrane to become much more permeable to sodium than to potassium. Hence, V_m would tend toward E_{Na} (about $+60$ mV), and an overshoot of zero potential would be expected. They predicted and showed, for example, that the action

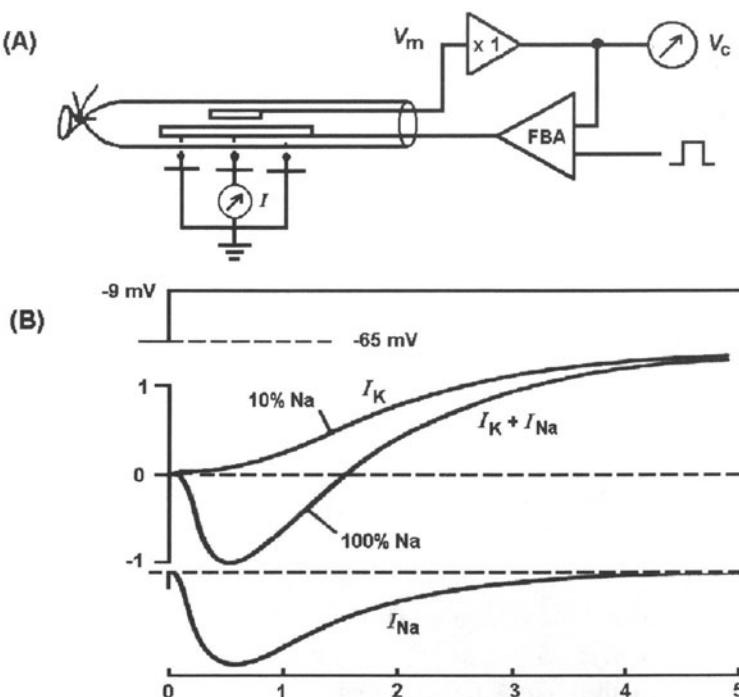


Figure 4.2 Quantitative measurements of ionic currents in the squid giant axon using space clamp and voltage clamp techniques. (A) Schematic of setup. Axial wire to impose space clamp. Feedback amplifier for voltage clamp delivers current to maintain membrane potential V_m at command level V_c (adapted from Hille 1984). (B) Current (mA/cm^2) versus time (msec). Low sodium concentration in bath eliminates I_{Na} , so that I_K is recorded. Then subtraction from total current in normal sodium yields I_{Na} . Clamping voltage is -9 mV , from a holding level of -65 mV . (Replotted data from Hodgkin and Huxley (1952a).)

potential amplitude depended critically on the concentration of external sodium; decreased sodium led to a lower peak for the action potential. In the theoretical section of their paper, they generalized the Nernst equation to predict the steady-state potential when the membrane is permeable with different degrees to more than one ionic species. This equation, modified from the earlier derivation by D. Goldman, is widely applied in cell biology, and is usually called the Goldman–Hodgkin–Katz equation.

The “sodium hypothesis” was a major conceptual advance. However, the question of how the permeability changes were dynamically linked to V_m was not completely addressed until the papers of 1952. Hodgkin and Huxley realized that by manipulating ionic concentrations in the axon and its environment, the contributions of different ionic conductances could be disentangled, provided that they responded independently to changes in V_m (a key assumption). By eliminating sodium from the bathing medium, I_{Na} becomes negligible and

so I_K is measured directly. Then I_{Na} can be determined by subtraction of I_K from the normal response as shown in Fig. 4.2B. Using this approach, Hodgkin and Huxley (1952a) were able to demonstrate convincingly that the current flowing across the squid axon membrane had only two major ionic components, I_{Na} and I_K , and that these currents were strongly influenced by V_m .

4.3 The Mathematical Model

Given this historical perspective, we can now better appreciate the insights provided by the HH model. In this section, we present the mathematical model itself. In subsequent sections we describe some of the experiments that Hodgkin and Huxley performed while developing their model, and we use GENESIS to simulate some of those experiments.

4.3.1 Electrical Equivalent Circuit

The HH model is based on the idea that the electrical properties of a segment of nerve membrane can be modeled by an equivalent circuit of the form shown in Fig. 4.3. In the equivalent circuit, current flow across the membrane has two major components, one associated with charging the membrane capacitance and one associated with the movement of specific types of ions across the membrane. The ionic current is further subdivided into three distinct components, a sodium current I_{Na} , a potassium current I_K , and a small leakage current I_L that is primarily carried by chloride ions.

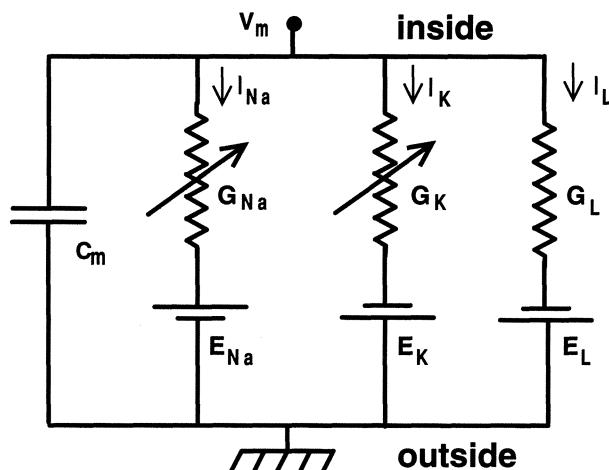


Figure 4.3 Electrical equivalent circuit proposed by Hodgkin and Huxley for a short segment of squid giant axon. The variable resistances represent voltage-dependent conductances (Hodgkin and Huxley 1952d).

The behavior of an electrical circuit of the type shown in Fig. 4.3 can be described by a differential equation of the form:

$$C_m \frac{dV_m}{dt} + I_{ion} = I_{ext}, \quad (4.1)$$

where C_m is the membrane capacitance, V_m is the intracellular potential (membrane potential), I_{ion} is the net ionic current flowing across the membrane, and I_{ext} is an externally applied current.

4.3.2 HH Conventions

Note that the appearance of I_{ion} on the left-hand side of Eq. 4.1 and I_{ext} on the right indicates that they have opposite *sign conventions*. As the equation is written, a positive external current I_{ext} will tend to depolarize the cell (i.e., make V_m more positive) whereas a positive ionic current I_{ion} will tend to hyperpolarize the cell (i.e., make V_m more negative). This sign convention for ionic currents is sometimes referred to as the *physiologists' convention* and is summarized by the phrase “inward negative,” meaning that an inward flow of positive ions into the cell is considered a negative current. This convention perhaps arose from the fact that when one studies an ionic current in a voltage clamp experiment, rather than measuring the ionic current directly, one actually measures the clamp current that is necessary to counterbalance it. Thus an inward flow of positive ions is observed as a negative-going clamp current, hence explaining the “inward negative” convention. While reading later chapters, it will be important to realize that internally GENESIS uses the opposite sign convention (“inward positive”), since that allows all currents to be treated consistently, without making a special case for ionic currents. In the *Squid* tutorial in this chapter, however, currents are plotted using the physiologists’ convention.

While we’re on the topic of *conventions*, there are two more issues that should be discussed here. The first concerns the *value* of the membrane potential V_m . Recall that potentials are relative; only potential differences can be measured directly. Thus when defining the intracellular potential V_m , one is free to choose a convention that defines the resting intracellular potential to be zero (the convention used by Hodgkin and Huxley), or one could choose a convention that defines the extracellular potential to be zero, in which case the resting intracellular potential would be around -70 mV . In either case the potential *difference* across the membrane is the same; it’s simply a matter of how “zero” is defined. GENESIS allows the user to choose any convention they like; in the *Squid* tutorial we use the HH convention that the resting membrane potential is zero.

The second convention we need to discuss concerns the *sign* of the membrane potential. The modern convention is that depolarization makes the membrane potential V_m more positive. However, Hodgkin and Huxley (1952d) use the opposite sign convention (depolarization negative) in their paper. In the *Squid* tutorial, we use the modern convention that

depolarization is positive. At a conceptual level, the choice of conventions for currents and potentials is inconsequential; however, at the implementation level it matters a great deal, since getting one of the signs wrong will cause the model to behave incorrectly. The most important thing in choosing conventions is to ensure that the choices are internally self-consistent. One must pay careful attention to these issues when implementing a GENESIS simulation using equations from a published model, since it may be necessary to convert the published equations into a form that is consistent with the rest of the simulation.

4.3.3 The Ionic Current

The total ionic current I_{ion} in Eq. 4.1 is the algebraic sum of the individual contributions from all participating ion types:

$$I_{ion} = \sum_k I_k = \sum_k G_k(V_m - E_k). \quad (4.2)$$

Each individual ionic component I_k has an associated conductance value G_k (conductance is the reciprocal of resistance, $G_k = 1/R_k$) and an equilibrium potential E_k (the potential for which the net ionic current flowing across the membrane is zero). The current is assumed to be proportional to the conductance times the driving force, resulting in terms of the general form $I_k = G_k(V_m - E_k)$. In the HH model of the squid giant axon, there are three such terms: a sodium current I_{Na} , a potassium current I_K , and a leakage current I_L :

$$I_{ion} = G_{Na}(V_m - E_{Na}) + G_K(V_m - E_K) + G_L(V_m - E_L). \quad (4.3)$$

In order to explain their experimental data, Hodgkin and Huxley postulated that G_{Na} and G_K changed dynamically as a function of membrane voltage. Today, we know that the basis for this voltage-dependence can be traced to the biophysical properties of the membrane channels that control the flow of ions across the membrane. It is important to remember that at the time Hodgkin and Huxley developed their model, there was very little information available about the biophysical structure of membrane or the molecular events underlying neural excitability. The modern concept of ion-selective membrane channels controlling the flow of ions across the membrane was only one of several competing ideas at the time. An important accomplishment of the HH model was to rule out several of the alternative ideas that had been proposed concerning membrane excitability.

Although Hodgkin and Huxley did not know about membrane channels when they developed their model, it is convenient for us to describe the voltage-dependent aspects of their model in those terms. The macroscopic conductances G_k of the HH model can be thought of as arising from the combined effects of a large number of microscopic ion channels embedded in the membrane. Each individual ion channel can be thought of as containing a small number of physical *gates* that regulate the flow of ions through the channel. An individual gate can be in one of two states, *permissive* or *non-permissive*. When *all* of the gates for

a particular channel are in the permissive state, ions can pass through the channel and the channel is *open*. If any of the gates are in the non-permissive state, ions cannot flow and the channel is *closed*.¹

The voltage-dependence of ionic conductances is incorporated into the HH model by assuming that the probability for an individual gate to be in the permissive or non-permissive state depends on the value of the membrane voltage. If we consider gates of a particular type i , we can define a probability p_i , ranging between 0 and 1, that represents the *probability* of an individual gate being in the permissive state. If we consider a large number of channels, rather than an individual channel, we can also interpret p_i as the *fraction* of gates in that population that are in the permissive state and $(1 - p_i)$ as the fraction in the non-permissive state. Transitions between permissive and non-permissive states in the HH model are assumed to obey first-order kinetics:

$$\frac{dp_i}{dt} = \alpha_i(V) (1 - p_i) - \beta_i(V) p_i, \quad (4.4)$$

where α_i and β_i are voltage-dependent *rate constants* describing the “non-permissive to permissive” and “permissive to non-permissive” transition rates, respectively. If the membrane voltage V_m is “clamped” at some fixed value V , then the fraction of gates in the permissive state will eventually reach a steady-state value (i.e., $dp_i/dt = 0$) as $t \rightarrow \infty$ given by:

$$p_{i,t \rightarrow \infty}(V) = \frac{\alpha_i(V)}{\alpha_i(V) + \beta_i(V)}. \quad (4.5)$$

The time course for approaching this equilibrium value is described by a simple exponential with time constant $\tau_i(V)$ given by:

$$\tau_i(V) = \frac{1}{\alpha_i(V) + \beta_i(V)}. \quad (4.6)$$

When an individual channel is open (i.e., when all the gates are in the permissive state), it contributes some small, fixed value to the total conductance and zero otherwise. The macroscopic conductance for a large population of channels is thus proportional to the number of channels in the open state which is, in turn, proportional to the probability that the associated gates are in their permissive state. Thus the macroscopic conductance G_k due to channels of type k , with constituent gates of type i , is proportional to the *product* of the individual gate probabilities p_i :

$$G_k = \bar{g}_k \prod_i p_i, \quad (4.7)$$

¹Although it would seem natural to speak of *gates* as being *open* or *closed*, a great deal of confusion can be avoided by consistently using the terminology *permissive* and *non-permissive* for *gates* while reserving the terms *open* and *closed* for *channels*.

where \bar{g}_k is a normalization constant that determines the maximum possible conductance when all the channels are open.

We have presented Eqs. 4.4–4.7 using a generalized notation that can be applied to a wide variety of conductances beyond those found in the squid axon. To conform to the standard notation of the HH model, the probability variable p_i in Eqs. 4.4–4.6 is replaced by a convenient notation in which the variable name is the same as the gate type. For example, Hodgkin and Huxley modeled the sodium conductance using three gates of a type labeled m and one gate of type h . Applying Eq. 4.7 to the sodium channel using both the generalized notation and the standard notation yields:

$$G_{Na} = \bar{g}_{Na} p_m^3 p_h \equiv \bar{g}_{Na} m^3 h. \quad (4.8)$$

Similarly, the potassium conductance is modeled with four identical “ n ” gates:

$$G_K = \bar{g}_K p_n^4 \equiv \bar{g}_K n^4. \quad (4.9)$$

Summarizing the ionic currents in the HH model in standard notation, we have:

$$I_{ion} = \bar{g}_{Na} m^3 h (V_m - E_{Na}) + \bar{g}_K n^4 (V_m - E_K) + \bar{g}_L (V_m - E_L), \quad (4.10)$$

$$\frac{dm}{dt} = \alpha_m(V) (1 - m) - \beta_m(V) m, \quad (4.11)$$

$$\frac{dh}{dt} = \alpha_h(V) (1 - h) - \beta_h(V) h, \quad (4.12)$$

$$\frac{dn}{dt} = \alpha_n(V) (1 - n) - \beta_n(V) n. \quad (4.13)$$

The task that remains is to specify exactly how the six rate constants in Eqs. 4.11–4.13 depend on the membrane voltage. Then Eqs. 4.10–4.13, together with Eq. 4.1, completely specify the behavior of the membrane potential V_m in the model.

4.4 Voltage Clamp Experiments

How did Hodgkin and Huxley go about determining the voltage-dependence of the rate constants α and β that appear in the kinetic equations Eqs. 4.11–4.13? How did they determine that the potassium conductance should be modeled with four identical n gates, but that the sodium conductance required three m gates and one h gate? In order to answer these questions, we need to look in some detail at the results of the voltage clamp experiments carried out by Hodgkin and Huxley.

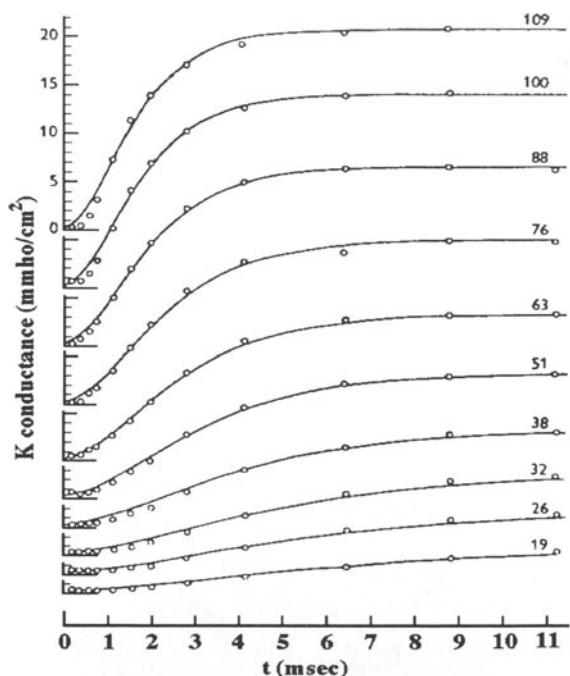


Figure 4.4 Experimental voltage clamp data illustrating voltage-dependent properties of the potassium conductance in squid giant axon. Data points are shown as open circles. Solid lines are best-fit curves of the form given in Eq. 4.20. The command voltage V_c (mV) is shown on the right-hand side of each curve. Redrawn from Hodgkin and Huxley (1952d).

4.4.1 Characterizing the K Conductance

Figure 4.4 shows some voltage clamp results obtained by Hodgkin and Huxley in which the time course of the potassium conductance is plotted for several different values of the command voltage. The most obvious trend in the data is that the steady-state K conductance level increases with increasing command voltage. A second, somewhat more subtle trend is that the rising phase of the conductance change becomes more rapid with increasing depolarization. For small depolarizations on the order of 20 mV , the half-maximum point occurs about 5 msec after the onset of the change in voltage, whereas for large depolarizations on the order of 100 mV , the half-maximum point is reached in about 2 msec .

Hodgkin and Huxley incorporated these voltage-dependent properties of the K conductance into a mathematical model by first writing down an equation that describes the time evolution of a first-order kinetic process:

$$\frac{dn}{dt} = \alpha_n(V)(1-n) - \beta_n(V)n. \quad (4.14)$$

In the experiments illustrated in Fig. 4.4, the membrane potential starts in the resting state ($V_m = 0$) and is then instantaneously stepped to a new clamp voltage V_c . What is the time course of the state variable n under these circumstances? Initially, with $V_m = 0$, the state variable n has a resting value given by Eq. 4.5,

$$n_\infty(0) = \frac{\alpha_n(0)}{\alpha_n(0) + \beta_n(0)}. \quad (4.15)$$

When V_m is clamped to V_c , n will eventually reach a steady-state value given by

$$n_\infty(V_c) = \frac{\alpha_n(V_c)}{\alpha_n(V_c) + \beta_n(V_c)}. \quad (4.16)$$

The solution to Eq. 4.14 that satisfies these boundary conditions is a simple exponential of the form

$$n(t) = n_\infty(V_c) - (n_\infty(V_c) - n_\infty(0))e^{-t/\tau_n}, \quad (4.17)$$

where

$$\tau_n(V_c) = \frac{1}{\alpha_n(V_c) + \beta_n(V_c)}. \quad (4.18)$$

Given Eq. 4.17, which describes the time course of n in response to a step change in command voltage, one could try fitting curves of this form to the conductance data shown in Fig. 4.4 by finding values of $n_\infty(0)$, $n_\infty(V_c)$, and $\tau_n(V_c)$ that give the best fit to the data for each value of V_c . Figure 4.5 illustrates this process, using some simulated conductance data generated by the Hodgkin–Huxley model. Recall that n takes on values between 0 and 1, so in order to fit the conductance data, n must be multiplied by a normalization constant \bar{g}_K that has units of conductance. For simplicity, the normalized conductance G_K/\bar{g}_K is plotted. The dotted line in Fig. 4.5 shows the best-fit results for a simple exponential curve of the form given in Eq. 4.17. Although this simple form does a reasonable job of capturing the general time course of the conductance change, it fails to reproduce the S-shaped (sigmoidal) trend in the data. This discrepancy is most apparent near the onset of the conductance change, shown in the inset of Fig. 4.5. Hodgkin and Huxley realized that a more sigmoidal time course could be generated if they considered the conductance to be proportional to a higher power of n . Figure 4.5 shows the results of fitting the conductance data using successively higher powers p . Using this sort of fitting procedure, Hodgkin and Huxley determined that a reasonable fit to their K conductance data could be obtained using a value of $p = 4$. Thus they arrived at a description for the K conductance given by

$$G_K = \bar{g}_K n^4, \quad (4.19)$$

in which case, the equation describing the conductance change and satisfying the appropriate boundary conditions is

$$G_K = \{(G_\infty(V_c))^{1/4} - ((G_\infty(V_c))^{1/4} - (G_\infty(0))^{1/4})e^{-t/\tau_n}\}^4, \quad (4.20)$$

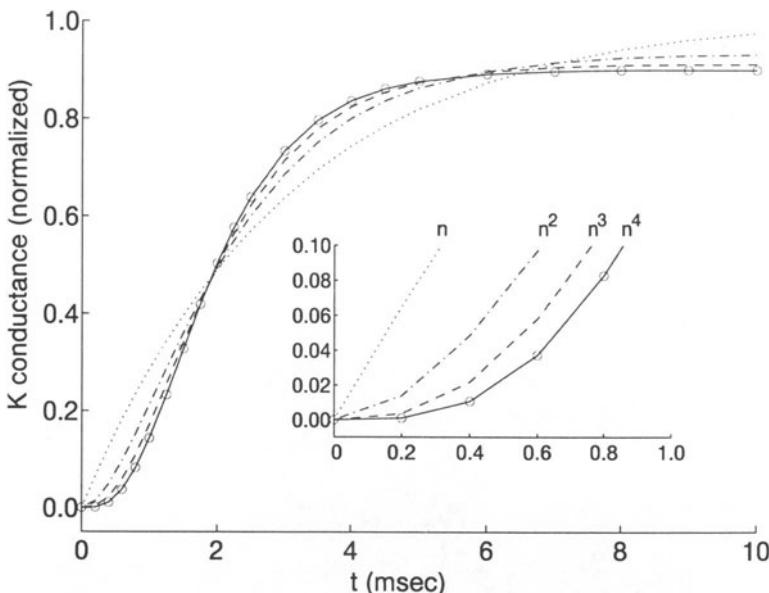


Figure 4.5 Best-fit curves of the form $G_k = \bar{g}_K n^p$ ($p = 1-4$) for simulated conductance vs. time data (open circles). The inset shows an enlargement of the first millisecond of the response. The initial inflection in the curve cannot be well fit by a simple exponential (dotted line) that rises linearly from zero. Successively higher powers of p ($p = 2$: dot-dashed; $p = 3$: dashed line) result in a better fit to the initial inflection. In this case, $p = 4$ (solid line) gives the best fit.

where $G_\infty(0)$ is the initial conductance and $G_\infty(V_c)$ is the steady-state conductance value attained when the command voltage is stepped to V_c . The solid lines in Fig. 4.4 show the best-fit results obtained by Hodgkin and Huxley for their data.

4.5 GENESIS: Voltage Clamp Experiments

In order to develop a better understanding of the procedures outlined above we will use the *Squid* tutorial to simulate a voltage clamp experiment of the type that Hodgkin and Huxley used to characterize the potassium conductance. Change to the *Scripts/squid* directory and start the *Squid* tutorial by typing “genesis Squid.” When the tutorial first loads, the simulation is in the current clamp mode. To switch to the voltage clamp mode, click the Toggle Vclamp/Iclamp Mode button. The first simulation experiment will be to hold the membrane potential at the resting potential for a couple of milliseconds (so we can see the baseline) and then rapidly clamp it to 50 mV above the resting potential. In this simulation, we measure voltages with respect to the resting potential, so we define the rest potential to be 0 volts, rather than -70 mV.

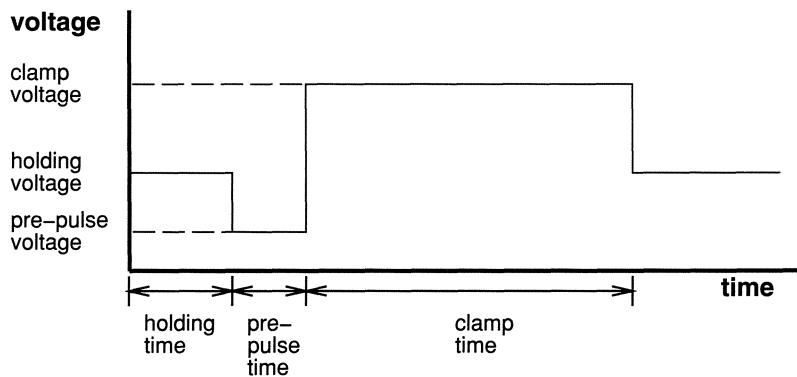


Figure 4.6 The time course of the voltage clamp signal. These quantities can be set in the Voltage Clamp Mode control panel.

Figure 4.6 defines the quantities that are set by the Voltage Clamp Mode control panel. Make sure they are set to the following:

Holding Voltage	= 0 mV
Holding Time	= 2 msec
Pre-pulse Voltage	= 0 mV
Pre-pulse Time	= 0 msec
Clamp Voltage	= 50 mV
Clamp Time	= 20 msec

Now run the simulation by clicking RESET followed by RUN. Your display should look similar to that shown in Fig. 4.7. The plot at the upper left shows both the command voltage which is applied to the voltage clamp circuitry and the resulting membrane potential. If everything is working properly, these two curves should be almost identical, since the idea of the voltage clamp is that the membrane voltage should exactly follow the command voltage. The lower left plot shows the injection current (clamp current) used to maintain the desired voltage. The time course of the clamp current has three components: a very brief positive-going spike at the onset of the voltage change related to the charging of the membrane capacitance, a transient negative (inward) current associated with the sodium conductance, and finally a sustained positive (outward) current associated with the potassium conductance. The quantities shown in the two left panels are experimental observables and were accessible to Hodgkin and Huxley in their experiments, whereas the two right panels on your screen show quantities that are not directly observable (channel conductances and channel currents), but which we can plot in the simulation by “peeking” at the internal state of the model. In thinking about the data that Hodgkin and Huxley had to work with, keep in mind that the observables are confined to the two left panels.

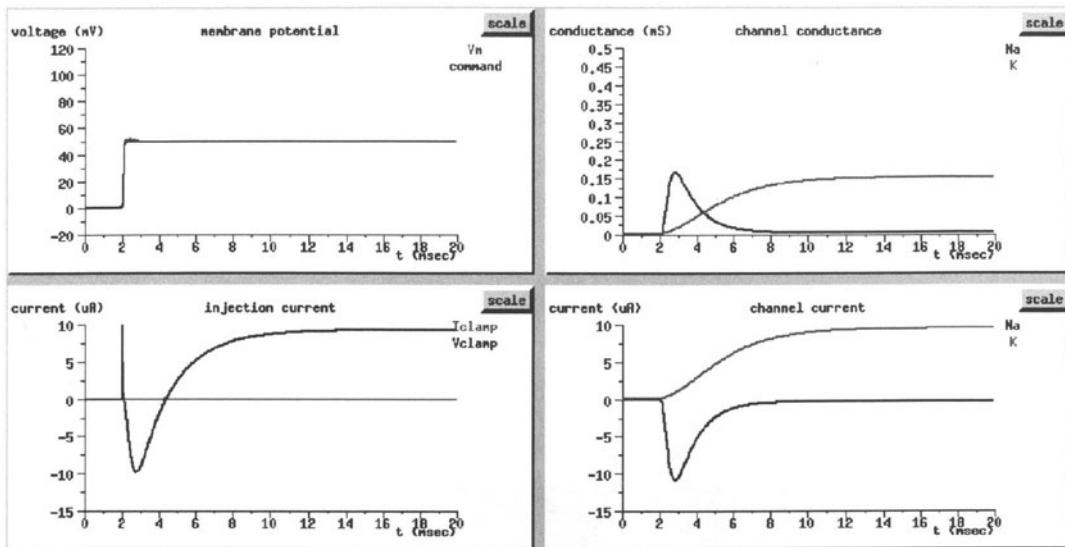


Figure 4.7 A voltage clamp experiment using the *Squid* tutorial. Upper left: membrane voltage and clamp voltage; lower left: clamp current; upper right: Na and K channel conductances; lower right: Na and K channel currents.

In order to study the K conductance alone, Hodgkin and Huxley replaced Na in the external bathing solution with an impermeant ion, thus eliminating most of the Na contribution to the measured currents. We have an even easier way of getting rid of the sodium current in the simulation: click the toggle button labeled “Na channel unblocked”, so that it reads “Na channel blocked.” Now rerun the simulation. Note that the clamp current no longer shows the transient negative (inward) current associated with the Na conductance.

Now let’s do a voltage clamp series to characterize the K conductance. We’ll start with a large voltage step (so we can adjust graph scales) and work our way down through a series of smaller values. Change the Clamp Voltage dialog box value to 100 mV and rerun the simulation. (Don’t forget to hit “Return” after changing the contents of the dialog box!) Notice that values in some of the plots go off scale. To correct this situation, we can use the **scale** buttons in the upper left-hand corner of each display. For example, to adjust the scale for the clamp current, click the **scale** button on the lower left graph, set **ymax** to 30, and then click **DONE**. If you want, you can adjust the scales of the other graphs in the same manner. Since we are going to perform a series of simulations and we want to see all the results plotted simultaneously, we’ll put the graphs into overlay mode (otherwise they get cleared on each reset). Click the toggle button labeled “Overlay OFF,” so that it reads “Overlay ON”. Now we’re ready to perform the next trial in the voltage series. Change the Clamp Voltage to 80 mV, click on **RESET**, and rerun the simulation. You should see the new data superimposed on the old data. Now continue the series with clamp voltages of 60,

40, and 20 mV. When you are finished, the right side of your display should look similar to Fig. 4.8.

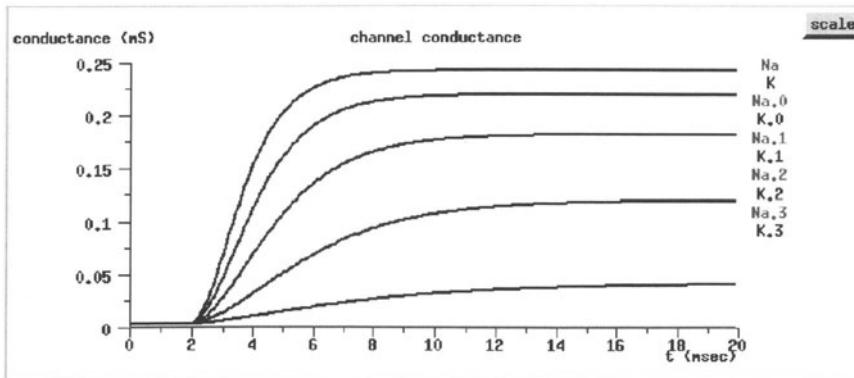


Figure 4.8 Plots of K conductance vs. time for a simulated voltage clamp series with Na channels blocked. Responses are shown for five values of the clamp voltage: 20, 40, 60, 80 and 100 mV. Compare with the experimental data in Fig. 4.4.

4.6 Parameterizing the Rate Constants

Using this procedure, Hodgkin and Huxley were able to determine the steady-state conductance values $n_\infty(V_c)$ and time constants $\tau_n(V_c)$ as a function of command voltage. Once values for $n_\infty(V_c)$ and $\tau_n(V_c)$ have been determined by fitting the conductance data, values for $\alpha_n(V_c)$ and $\beta_n(V_c)$ can be found from the following relationships:

$$\alpha_n(V) = \frac{n_\infty(V)}{\tau_n(V)} \quad (4.21)$$

$$\beta_n(V) = \frac{1 - n_\infty(V)}{\tau_n(V)}. \quad (4.22)$$

The open circles in Fig. 4.9 represent the experimentally determined values of $n_\infty(V_c)$, $\tau_n(V_c)$, $\alpha_n(V_c)$, and $\beta_n(V_c)$ as a function of command voltage. Hodgkin and Huxley then found smooth curves that went through these data points. The empirically determined expressions for the rate constants α_n and β_n are:

$$\alpha_n(V) = \frac{0.01(10 - V)}{\exp(\frac{10 - V}{10}) - 1} \quad (4.23)$$

$$\beta_n(V) = 0.125 \exp(-V/80). \quad (4.24)$$

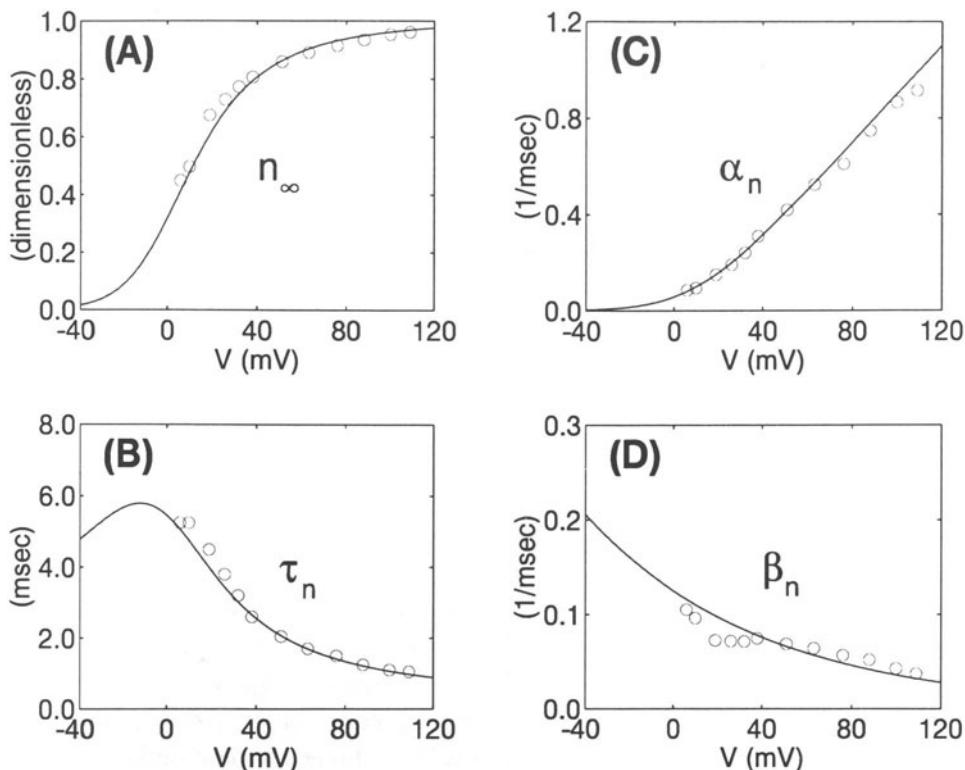


Figure 4.9 Voltage dependence of K conductance parameters in the HH model. (A) Steady-state value $n_\infty(V)$; (B) time constant $\tau_n(V)$; (C) rate constant $\alpha(V)$; and (D) rate constant $\beta(V)$. Open circles in (A) and (B) are best-fit parameters from voltage clamp data of the type shown in Fig. 4.4. Open circles in (C) and (D) are computed from Eqs. 4.21–4.22. Solid lines in (C) and (D) are empirical fits to the rate constant data of the form given in Eqs. 4.23–4.24. Solid lines in (A) and (B) are then calculated from Eqs. 4.16 and 4.18.

If you compare these expressions with Eqs. 12–13 in Hodgkin and Huxley (1952d), you will note that the sign of the membrane voltage has been changed to correspond to the modern convention (see Sec. 4.3.2).

4.7 Inactivation of the Na Conductance

There is an important qualitative difference between the Na and the K conductance changes that are observed in the squid axon under voltage clamp conditions. Namely, in response to a sustained voltage clamp step, the change in Na conductance is transient and only lasts a few milliseconds, whereas the change in K conductance is sustained and lasts as long as the voltage clamp is maintained. This effect can be seen in the upper right panel of Fig. 4.7.

To explore the Na conductance change in more detail, run the *Squid* simulation in voltage clamp mode with Na channels unblocked and K channels blocked (use the toggle buttons on the control form). Set the maximum simulation time to 10 msec in the Simulation Control panel and set the following values in the Voltage Clamp Mode control panel dialog boxes:

Holding Voltage	= 0 mV
Holding Time	= 2 msec
Pre-pulse Voltage	= 0 mV
Pre-pulse Time	= 0 msec
Clamp Voltage	= 50 mV
Clamp Time	= 8 msec

Place the graphs in overlay mode (use the Overlay ON toggle button in the Simulation Control panel), and observe the magnitude and time course of the Na conductance change for clamp voltages between 10 and 70 mV. You will note that the magnitude of the peak conductance increases with increasing voltage and that the time constants of the rising and falling phases generally become shorter (faster) with increasing voltage.

In order to model these transient conductance changes, Hodgkin and Huxley needed to use a system of differential equations that was at least second-order. Using the same strategy as for the K conductance, they chose to do this by building up the higher-order response dynamics using a set of variables such that each obeys first-order kinetics. To describe the activation and inactivation phases of the Na conductance change requires two separate state variables, which Hodgkin and Huxley labeled m (the activation variable) and h (inactivation). In order to accurately capture the initial inflection in the conductance change, they found that they had to raise the m variable to the third power. Thus they arrived at a description for the Na conductance given by

$$G_{Na} = \bar{g}_{Na} m^3 h. \quad (4.25)$$

Following a procedure very similar to that outlined previously for the K conductance, Hodgkin and Huxley determined the voltage-dependence of the rate constants that govern the activation and inactivation variables. The empirically determined expressions that they arrived at for describing α_m , β_m , α_h and β_h are:

$$\alpha_m(V) = \frac{0.1(25 - V)}{\exp(\frac{25-V}{10}) - 1}, \quad (4.26)$$

$$\beta_m(V) = 4 \exp(-V/18), \quad (4.27)$$

$$\alpha_h(V) = 0.07 \exp(-V/20), \quad (4.28)$$

$$\beta_h(V) = \frac{1}{\exp(\frac{30-V}{10}) + 1}. \quad (4.29)$$

Again, to be consistent with the modern sign convention used in GENESIS, we have flipped the sign of the voltage relative to the original Eqs. 20, 21, 23, 24 in Hodgkin and Huxley (1952d).

4.8 Current Injection Experiments

As we have seen, the form of the HH model and the values of the model parameters were all empirically determined from voltage clamp data. There was no *a priori* guarantee that the model would necessarily be successful in predicting the behavior of the squid axon under other experimental conditions. Thus it must have been extraordinarily satisfying for Hodgkin and Huxley to see their model produce realistic-looking action potentials when they numerically simulated the response to superthreshold current injections.

Run the *Squid* simulation in current clamp mode. Make sure that both the Na and K channels are unblocked. Set the maximum simulation time to 50 msec and set the following values in the Current Clamp Mode control panel:

Base Current	= 0.0 uA
Pulse Current 1	= 0.1 uA
Onset Delay 1	= 5.0 msec
Pulse Width 1	= 30.0 msec
Pulse Current 2	= 0.0 uA
Onset Delay 2	= 0.0 msec
Pulse Width 2	= 0.0 msec
Pulse Mode	= Single Pulse

With this set of parameters, you should observe a short train of action potentials in response to the injected current. Section 4.9 provides several suggested exercises for exploring properties of the HH model under current clamp conditions, including threshold behavior, refractory periods, depolarization block and anode break excitation.

4.9 Exercises

1. In voltage clamp mode, generate plots of peak conductance versus clamp voltage and peak current versus clamp voltage for the Na and K currents. (Characterize the Na and K components individually by using the appropriate toggle button to block the other component.) Select clamp voltages that cover the range from 40 mV below resting potential to 140 mV above resting potential. For each case, determine the

peak conductance and current from the graphs and use them in your plots. What is the general shape of the conductance vs. voltage plots? What is the general shape of the current vs. voltage plots? What are the reversal potentials for Na and K?

2. In voltage clamp mode, examine the effect of giving different hyperpolarizing pre-conditioning pulses prior to the voltage clamp step. (Suggested parameters: holding voltage = 0 mV ; holding time = 5 msec ; pre-pulse voltage = 0 to -50 mV in 10 mV steps; pre-pulse time = 5 msec ; clamp voltage = $+40\text{ mV}$; clamp time = 20 msec .) What is the effect of the pre-conditioning pulse on the Na conductance? On the K conductance? In the context of the HH model, describe the mechanism responsible for this effect. How might this relate to the “after-hyperpolarization” that follows an action potential?
3. In current clamp mode, find the minimum current (threshold current) for eliciting a single action potential. (Suggested settings: base current = $0\text{ }\mu\text{A}$; onset delay 1 = 5 msec ; pulse width 1 = 15 msec ; simulation time = 20 msec .) How “sharp” is the threshold phenomenon — can you find a value of the injected current that gives a “half-height” action potential? If the threshold appears to be “all-or-none,” report the minimum fractional change in injection current that you tested (e.g., 1 part in 100, 1-in-1000, etc.).
4. The *rheobase* current is the minimum current that will elicit repetitive firing (i.e., generate a train of action potentials). What is the rheobase current for the *Squid* model? (Suggested settings: base current = $0\text{ }\mu\text{A}$; onset delay 1 = 5 msec ; pulse width 1 = 95 msec ; simulation time = 100 msec .) How sharp is the transition from single spike generation to repetitive firing? — Can you find a value of the injected current that generates *two* action potentials, but doesn’t fire repetitively?
5. By counting the number of spikes generated in a 100 msec window, construct a plot of firing frequency vs. injected current, starting at the rheobase current and working up to a value of about 10 times rheobase. (Suggested settings: base current = $0\text{ }\mu\text{A}$; onset delay 1 = 0 msec ; pulse width 1 = 100 msec ; simulation time = 100 msec .) How much does a 10-fold increase in injected current increase the firing rate? What happens if you increase the injected current to 100 times rheobase?
6. In problem 3 we saw that single action potentials can be elicited by small *sustained* levels of current injection. Single action potentials can also be elicited by *transient* pulses of current injection, even when the duration of the pulse is shorter than the duration of the action potential. As the length of the pulse decreases, however, the amplitude necessary to elicit an action potential increases. Generate a plot of single spike threshold current vs. pulse duration for pulse widths between 0.1 and 2.0 msec .

Is there a simple relationship between pulse width and threshold current? (Use an integration time step of 0.01 msec for this study.)

7. In this problem you will investigate the *refractory period* that follows each action potential. The *absolute* refractory period is the time interval during which no stimulus, regardless of strength, is capable of generating another action potential in the axon. The *relative* refractory period is the time interval during which a second action potential can be generated, but which requires an increased stimulus amplitude in order to do so. Using the two-pulse capability of the current clamp mode, map out the absolute and relative refractory periods of the model by generating a plot of threshold amplitude vs. latency. Use a pulse width of 1 msec. How long is the absolute refractory period? The relative refractory period? (When mapping out the absolute refractory period, make sure that the responses you call “spikes” are true “all-or-none” phenomena.)
8. All of the injection pulses in the previous current clamp problems have been depolarizing. In this problem you will look at the effect of hyperpolarizing current pulses. Set the pulse amplitude to $-0.1 \mu A$ and set the pulse duration to 5 msec. What happens? What is the threshold, in terms of current magnitude and pulse duration, for eliciting this so-called *anode break* excitation? What mechanisms in the model are responsible for this behavior? (Hint: look at the time course of the state variables m , n and h , using the State Plot Visible toggle button.)
9. Set the base current level just above rheobase to establish repetitive firing. Now superimpose a 1 msec duration, $0.1 \mu A$ current pulse at various latencies ranging from 5.0 to 15.0 msec. Can you find a latency value that abolishes the repetitive firing?

Chapter 5

Cable and Compartmental Models of Dendritic Trees

IDAN SEGEV

5.1 Introduction

In the previous chapter, we used a single compartment model to study the mechanisms for the activation of voltage-activated channels, which produce neuron firing. Next, we need to understand how inputs to the neuron affect the potential in the soma and other regions that contain these channels. The following chapter deals with the response of the neuron to synaptic inputs to produce postsynaptic potentials (PSPs). In this chapter, we concentrate on modeling the spread of the PSP through the dendritic tree.

Dendrites are strikingly exquisite and unique structures. They are the largest component in both surface area and volume of the brain and their specific morphology is used to classify neurons into classes: pyramidal, Purkinje, amacrine, stellate, etc. (Fig. 5.1). But most meaningful is that the majority of the synaptic information is conveyed onto the dendritic tree and it is there where this information is processed. Indeed, dendrites are the elementary computing device of the brain.

A typical dendritic tree receives approximately ten thousand synaptic inputs distributed over the dendritic surface. When activated, each of these inputs produces a local conductance change for specific ions at the postsynaptic membrane, followed by a flow of the corresponding ion current between the two sides of the postsynaptic membrane. As a result, a local change in membrane potential is generated and then spreads along the dendritic branches.

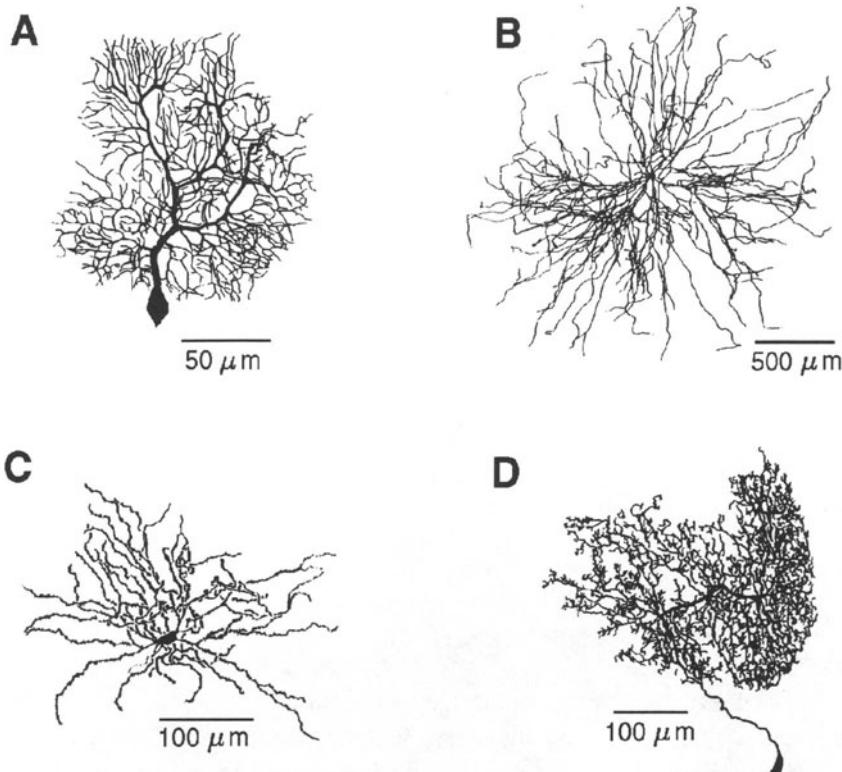


Figure 5.1 Dendrites have unique shapes which are used to characterize neurons into types. (A) Cerebellar Purkinje cell of the guinea pig (reconstructed by Moshe Rapp), (B) α -motoneuron from the cat spinal cord (reconstructed by Robert Burke), (C) Neostratal spiny neuron from the rat (Wilson 1992), (D) Axonless interneurons of the locust (reconstructed by Giles Laurent). In many neuron types, synaptic inputs from a given source are preferentially mapped into a particular region of the dendritic tree. For example, in Purkinje cells, one excitatory input comes from the vast number of synapses (more than 100,000) that specifically contact spines on the thin tertiary branches, whereas the other excitatory input comes from the climbing fiber that contacts the thick (smooth) dendrites. Inhibition from the basket cells impinges close to the Purkinje cell soma, whereas inhibition from stellate cells contacts mainly distal parts of the tree. Note the differences in scales for the different neuron types.

How does this spread depend on the morphology (the branching pattern) of the tree and on the electrical properties of its membrane and cytoplasm? This question is a fundamental one; its answer will provide the understanding of how the various synaptic inputs that are distributed over the dendritic tree interact in time and in space to determine the input-output properties of the neuron and, consequently, their effect upon the computational capability of the neuronal networks that they constitute.

Cable theory for dendrites was developed in 1959 by W. Rall precisely for this purpose. Namely, to derive a mathematical model that describes the flow of electric current (and the spread of the resultant voltage) in morphologically and physiologically realistic dendritic trees that receive synaptic inputs at various sites and times. In the last thirty years, cable theory for dendrites, complemented by the compartmental modeling approach (Rall 1964), played an essential role in the estimation of dendritic parameters, in designing and interpreting experiments and in providing insights into the computational function of dendrites. This chapter attempts to briefly summarize cable theory and compartmental models, and to highlight the main results and insights obtained from applying cable and compartmental models to various neuron types. A complete account of Rall's studies, including his principal papers, can be found in Segev, Rinzel and Shepherd (1995).

We begin by getting acquainted with dendrites, the subject matter of this chapter. We then introduce the concepts and assumptions that led to the development of cable theory and then introduce the cable equation. Next, we discuss the implication of this equation for several important theoretical cases as well as for the fully reconstructed dendritic tree. The numerical solution of the cable equation using compartmental techniques is then presented. We summarize the chapter with the main insights that were gained from implementing cable and compartmental models with neurons of various types. Along the way, and at the end of the chapter, we suggest several exercises using GENESIS that will help the reader to understand the principles that govern signal processing in dendritic trees.

5.2 Background

5.2.1 Dendritic Trees: Anatomy, Physiology and Synaptology

Following the light microscopic studies of the phenomenal neuro-anatomist, Ramón y Cajal, dendrites became the focus of many anatomical investigations and today, with the aid of electron micrograph studies and computer-driven reconstructing techniques, we have a rather intimate knowledge of the fine structure of dendrites. These studies allowed us to obtain essential information on the exact site and type (excitatory or inhibitory) of synaptic inputs as well as on the dimensions of dendrites including the fine structures, the dendritic spines, that are involved in the synaptic processing (White 1989, Shepherd 1990). Here we attempt to introduce, in a concise way, some facts about dendrites and their synaptic inputs. One should remember that, because dendrites come in many shapes and sizes, such a summary

unavoidably gives only a rough range of values; more information can be found in Segev (1995).

Branching

Dendrites tend to bifurcate repeatedly and create (often several) large and complicated trees. Cerebellar Purkinje cells, for example, typically bear only one very complicated tree with approximately 400 terminal tips (Fig. 5.1A), whereas α -motoneurons from the cat spinal cord typically possesses 8–12 trees; each has approximately 30 terminal tips (Fig. 5.1B). The dendrites of each type of neurons have a unique branching pattern that can be easily identified and thus help to classify neurons.

Diameters

Dendrites are thin tubes of nerve membrane. Near the soma they start with a diameter of a few μm ; their diameter typically falls below 1 μm as they successively branch. Many types of dendrites (e.g., cerebellar Purkinje cells, cortical and hippocampal pyramidal) are studded with abundant tiny appendages, the dendritic spines, which create very thin ($\sim 0.1 \mu m$) and short ($\sim 1 \mu m$) dendritic branches. When present, dendritic spines are the major postsynaptic target for excitatory synaptic inputs and they seem to be important loci for plastic processes in the nervous system (Rall 1974 in Segev et al. 1995, Segev and Rall 1988, Koch and Zador 1993).

Length

Dendritic trees may range from very short (100–200 μm , as in the spiny stellate cell in the mammalian cortex) to quite long (1–2 mm , for the spinal α -motoneurons). The total dendritic length may reach $10^4 \mu m$ (1 cm) and more.

Area and Volume

As mentioned in the introduction, the majority of the brain volume and area is occupied by dendrites. The surface area of a single dendritic tree is in the range of 2,000–750,000 μm^2 ; its volume may reach up to 30,000 μm^3 .

Physiology of Dendrites

Both the intracellular cytoplasmic core and the extracellular fluid of dendrites are composed of ionic media that can conduct electric current. The membrane of dendrites can also conduct current via specific transmembrane ion channels, but the resistance to current flow across the membrane is much greater than the resistance along the core. In addition to the

membrane channels (membrane resistance), the dendritic membrane can store ionic charges, thus behaving like a capacitor. These R-C properties of the membrane imply a time constant ($\tau_m = RC$) for charging and discharging the transmembrane voltage. The typical range of values for τ_m is 1–100 msec. Also, the membrane and cytoplasm resistivity imply an input resistance (R_{in}) at any given point in the dendritic tree. R_{in} can range from 1 M Ω (at thick and leaky dendrites) and can reach 1000 M Ω (at thin processes, such as dendritic spines). The large values of R_{in} expected in dendrites imply that small excitatory synaptic conductance change (of ~ 1 nS) will produce, locally, a significant (a few tens of mV) voltage change. More details of the biophysics of dendrites, the specific properties of their membrane and cytoplasm and their electrical (rather than anatomical) length are considered below.

In classical cable theory, the assumption was that the electrical properties of the membrane and cytoplasmic properties are *passive* (voltage-independent) so that one could correctly speak of a membrane time *constant* and of a fixed input resistance (at a given site of the dendritic tree). However, recent recordings from dendrites (e.g., Stuart and Sakmann 1994) clearly demonstrate that the dendritic membrane of many neurons is endowed with voltage-gated ion channels. This complicates the situation (and makes it more interesting) since, now, the membrane resistivity (and thus τ_m and R_{in}) are voltage-dependent. For sufficiently large voltage perturbations, this nonlinearity may have important consequences on dendritic processing (Rall and Segev 1987). This important issue is further discussed below.

Synaptic Types and Distribution

Synapses are not randomly distributed over the dendritic surface. In general, inhibitory synapses are more proximal than excitatory synapses, although they are also present at distal dendritic regions and, when present, on some spines in conjunction with an excitatory input. In many systems (e.g., pyramidal hippocampal cells and cerebellar Purkinje cells), a given input source is preferentially mapped onto a given region of the dendritic tree (Shepherd 1990), rather being randomly distributed over the dendritic surface.

The time course of the synaptic conductance change associated with the various types of inputs in a given neuron may vary by 1–2 orders of magnitude. The fast excitatory (AMPA or non-NMDA) and inhibitory (GABA_A) inputs operate on a time scale of 1 msec and have a peak conductance on the order of 1 nS; this conductance is approximately 10 times larger than the slow excitatory (NMDA) and inhibitory (GABA_B) inputs that both act on a slower time scale of 10–100 msec.

5.2.2 Summary

Dendrites and their spines are the target for a large number of synaptic inputs that, in many cases, are non-randomly distributed over the dendritic surface. The dendritic membrane

is equipped with a variety of synaptically activated and voltage-gated ion channels. The kinetics and voltage dependence of these channels, together with a particular dendritic morphology and input distribution, make the dendritic tree behave as a complex *dynamical* device with a potentially rich repertoire of computational (input-output) capabilities. Cable theory for dendrites provides the mathematical framework that enables one to connect the morphological and electrical structure of the neuron to its function.

5.3 The One-Dimensional Cable Equation

5.3.1 Basic Concepts and Assumptions

As mentioned previously, dendrites are thin tubes wrapped with a membrane that is a relatively good electrical insulator compared to the resistance provided by the intracellular core or the extracellular fluid. Because of this difference in membrane vs. axial resistivity, for a short length of dendrite, the electrical current inside the core conductor tends to flow parallel to the cylinder axis (along the x -axis). This is why the classical cable theory considers only one spatial dimension (x) along the cable, while neglecting the y and z dimensions. In other words, one key assumption of the one-dimensional cable theory is that the voltage V across the membrane is a function of only time t and distance x along the core conductor.

The other fundamental assumptions in the classical cable theory are: (1) The membrane is passive (voltage-independent) and uniform. (2) The core conductor has constant cross section and the intracellular fluid can be represented as an ohmic resistance. (3) The extracellular resistivity is negligible (implying extracellular isopotentiality). (4) The inputs are currents (which sum linearly, in contrast to changes in synaptic membrane conductance, whose effects do not sum linearly, as we shall see in Chapter 6). For convenience, we will make the additional assumption that the membrane potential is measured with respect to a resting potential of zero, as we assumed in the previous chapter.

These assumptions allow us to write down the one-dimensional passive cable equation for $V(x, t)$, the voltage across the membrane cylinder at any point x and time t . As was shown by Rall (1959), this equation can be solved analytically for arbitrarily complicated passive dendritic trees. As noted before, real dendritic trees receive conductance inputs (not current inputs) and may possess nonlinear membrane channels (violating the passive assumption). Yet, as we discuss later, the passive case is very important as the essential reference case, and it provided the fundamental insights regarding signal processing in dendrites.

5.3.2 The Cable Equation

At any point along a cylindrical membrane segment (core conductor), current can flow either longitudinally (along the x -axis), or through the membrane. The longitudinal current I_i (in amperes) encounters the cytoplasm resistance, producing a voltage drop. We take this

current to be positive when flowing in the direction of increasing values of x , and define the *cytoplasm resistivity* as a resistance per unit length along the x -axis r_i , expressed in units of Ω/cm . Then, Ohm's law allows us to write

$$\frac{1}{r_i} \frac{\partial V}{\partial x} = -I_i. \quad (5.1)$$

The membrane current can either cross the membrane via the passive (resting) membrane channels, represented as a resistance r_m (in $\Omega \cdot cm$) for a unit length, or charge (discharge) the membrane capacitance per unit length c_m (in F/cm). If no additional current is applied from an electrode, then the change per unit length ($\partial I_i / \partial x$) of the longitudinal current must be the density of the membrane current i_m per unit length (taken positive outward),

$$\frac{\partial I_i}{\partial x} = -i_m = -\left(\frac{V}{r_m} + c_m \frac{\partial V}{\partial t}\right). \quad (5.2)$$

Combining Eq. 5.1 and Eq. 5.2, we get the cable equation, a second-order partial differential equation (PDE) for $V(x, t)$,

$$\frac{1}{r_i} \frac{\partial^2 V}{\partial x^2} = c_m \frac{\partial V}{\partial t} + \frac{V}{r_m}. \quad (5.3)$$

For the derivation of Eq. 5.3, it has been useful to consider the cytoplasm resistivity r_i , membrane resistivity r_m and membrane capacitance c_m for a unit length of cable having some fixed diameter. If we want to describe the cable properties in terms of the cable diameter, or we wish to make a compartmental model of a dendrite based on short sections of length l (Sec. 5.5), we will need expressions for the actual resistances and capacitance in terms of the cable dimensions.

It is often useful to refer to the membrane capacitance or resistance of a patch of membrane that has an area of 1 cm^2 , so that our calculations can be independent of the size of a neural compartment. These quantities are called the *specific capacitance* and *specific resistance* of the membrane. In this book, and in the GENESIS tutorials, we denote the specific capacitance by C_M and the specific resistance by R_M , and use the symbols C_m and R_m for the actual values of the membrane capacitance and resistance of a section of dendritic cable in farads and ohms. This can be a point of confusion when reading other descriptions of cable theory, as it is also common to use the same notation (C_m and R_m) for the specific quantities.

The capacitance of biological membranes was found to have a specific value C_M close to $1 \mu F/cm^2$. Hence, the actual capacitance C_m of a patch of cylindrical membrane with diameter d and length l is $\pi dl C_M$. In terms of the capacitance per unit length, $C_m = l c_m$. If the passive channels are uniformly distributed over a small patch of membrane, the conductance will be proportional to the membrane area. This means that the membrane resistance will be inversely proportional to the area and that it can be written as $R_m = R_M / (\pi dl)$, or as

$R_m = r_m/l$. R_M is then expressed in units of $\Omega \cdot \text{cm}^2$. Later in this chapter, we perform some simulations in which a number of cylindrical compartments are connected through their axial resistances R_a . As this resistance is proportional to the length of the compartment and inversely proportional to its cross sectional area, we can define a *specific axial resistance* R_A that is independent of the dimensions of the compartment and has units of $\Omega \cdot \text{cm}$. Thus, a cylindrical segment of length l and diameter d will have an axial resistance R_a of $(4lR_A)/\pi d^2$, or lr_i .

We can summarize these relationships with the equations

$$C_m = c_m l = \pi dl C_M, \quad (5.4)$$

$$R_m = r_m/l = \frac{R_M}{\pi dl} \quad (5.5)$$

and

$$R_a = r_i l = \frac{4lR_A}{\pi d^2}. \quad (5.6)$$

It is useful to define the *space constant*,

$$\lambda = \sqrt{r_m/r_i} = \sqrt{(d/4)R_M/R_A} \quad (5.7)$$

(in cm) and the *membrane time constant*,

$$\tau_m = r_m c_m = R_M C_M = R_m C_m. \quad (5.8)$$

Then, the cable equation (Eq. 5.3) becomes

$$\lambda^2 \frac{\partial^2 V}{\partial x^2} - \tau_m \frac{\partial V}{\partial t} - V = 0, \quad (5.9)$$

or in dimensionless units,

$$\frac{\partial^2 V}{\partial X^2} - \frac{\partial V}{\partial T} - V = 0, \quad (5.10)$$

where $X = x/\lambda$ and $T = t/\tau_m$. A complete derivation of the cable equation can be found in Rall (1989) and in Jack, Noble and Tsien (1975).

5.4 Solution of the Cable Equation for Several Cases

5.4.1 Steady-State Voltage Attenuation with Distance

The solution of Eq. 5.10 depends, in addition to the electrical properties of the membrane and cytoplasm, on the initial condition and the boundary condition at the end of the segment toward which the current flows. Consider the simple case of a steady state ($\partial V / \partial t = 0$); the cable equation (5.10) is reduced to an ordinary differential equation.

$$\frac{d^2 V}{d X^2} - V = 0, \quad (5.11)$$

whose general solution can be expressed as

$$V(X) = A e^X + B e^{-X}, \quad (5.12)$$

where A and B depend on the boundary conditions. In the case of a cylindrical segment of infinite extension, where $V = 0$ at $X = \infty$, and $V = V_0$ at $X = 0$, the solution for Eq. 5.11 is

$$V(X) = V_0 e^{-X} = V_0 e^{-x/\lambda}. \quad (5.13)$$

Thus, in this case, the steady voltage attenuates exponentially with distance. Indeed, in a very long uniform cylindrical segment, a steady voltage attenuates e -fold for each unit of λ . This holds only for a cylinder of infinite length.

Now, let us consider a finite length of dendritic cable. If it has a length l , we can define the dimensionless *electrotonic length* as $L = l/\lambda$. When the cylindrical segment has a sealed end at $X = L$ (“open circuit termination”), no longitudinal current flows at this end. Then, the solution for Eq. 5.11 with $V = V_0$ at $X = 0$ is

$$V(X) = \frac{V_0 \cosh(L - X)}{\cosh(L)}, \text{ for } \frac{\partial V}{\partial X} = 0 \text{ at } X = L. \quad (5.14)$$

As shown in Fig. 5.2, in finite cylinders with sealed ends, steady voltage attenuates less steeply than $\exp(-x/\lambda)$. In the other extreme, where the point $X = L$ is clamped to the resting potential (denoted here, for simplicity, as 0), the solution of Eq. 5.11 is

$$V(X) = \frac{V_0 \sinh(L - X)}{\sinh(L)}, \text{ for } V = 0 \text{ at } X = L. \quad (5.15)$$

In this case, steady voltage attenuates more steeply than the attenuation in the infinite case (Fig. 5.2). In dendritic trees, a dendritic segment typically ends with a subtree; this “leaky end” condition is somewhere between the sealed end (Eq. 5.14) and the “clamped to rest” condition (Eq. 5.15). (See Sec. 5.4.5.)

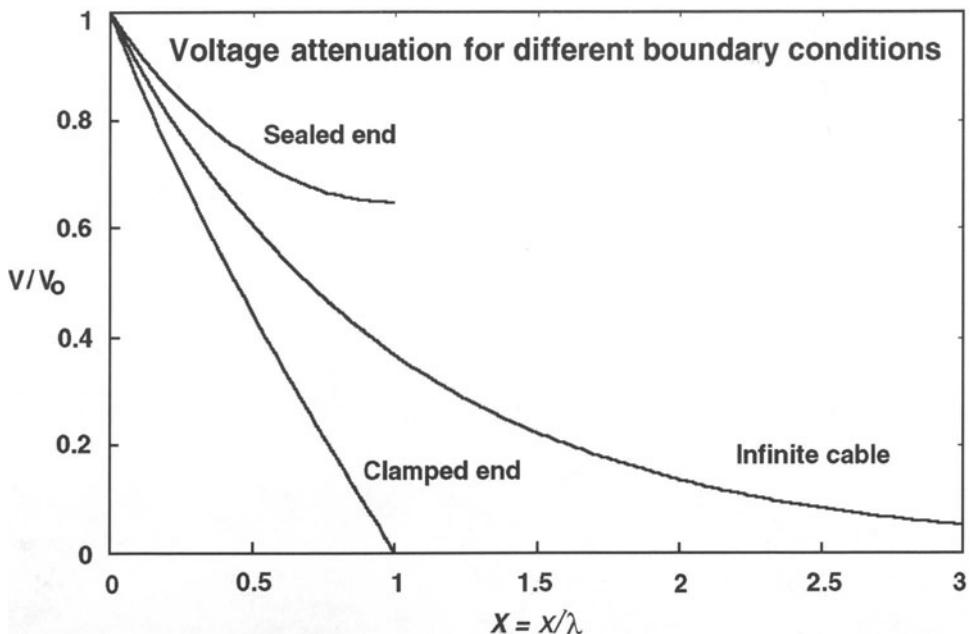


Figure 5.2 Attenuation of membrane potential with distance in a cylindrical cable with different boundary conditions. The middle curve shows the attenuation in an infinite cable (Eq. 5.13). The other two plots are for a finite length cable of electrotonic length $L = 1$. The upper one is for a sealed end cable (no current flows past the end, Eq. 5.14) and the lower is for a cable with the end clamped to the resting potential of 0 (Eq. 5.15).

5.4.2 Voltage Decay with Time

Consider the other extreme case of the cable equation (5.10) where $\partial V / \partial x = 0$. The cable is “shrunken” to an isopotential element, and Eq. 5.10 is reduced to an ordinary differential equation,

$$\frac{dV}{dT} + V = 0, \quad (5.16)$$

whose general solution can be expressed as

$$V(T) = Ae^{-T}, \quad (5.17)$$

where A depends on the initial condition. When a current step I_{pulse} is injected to this isopotential neuron, the resultant voltage V is

$$V(T) = I_{pulse}R_m(1 - e^{-T}) = I_{pulse}R_m(1 - e^{-t/\tau_m}), \quad (5.18)$$

where R_m is the net membrane resistance (in ohms) of this isopotential segment. At the cessation of the current at $t = t_0$, the voltage decays exponentially from its maximal value

$$V_0 = V(t_0),$$

$$V(T) = V_0 e^{-T} = V_0 e^{-t/\tau_m}, \text{ for } t \geq t_0. \quad (5.19)$$

Equation 5.18 implies that, because of the R-C properties of the membrane, the voltage developed as a result of the current input lags behind the current input; Equation 5.19 implies that voltage remains for some time after the input ends (“memory”). This situation is discussed in more detail in Chapter 6.

In the general case of passive cylinders, the solution to the cable equation (Eq. 5.10) can be expressed as a sum of an infinite number of exponential decays,

$$V(X, T) = \sum_{i=0}^{\infty} B_i \cos(i\pi X/L) e^{-t/\tau_i}, \quad (5.20)$$

where the Fourier coefficients B_i depend only on L , the index i , and on the initial conditions (the input point and the initial distribution of voltage over the tree). The time constants τ_i are independent of location in the tree; $\tau_i < \tau_{i+1}$ for any i and, for the uniform membrane, the slowest time constant $\tau_0 = \tau_m$. The shorter (“equalizing”) time constants (τ_i , for $i = 1, 2, \dots$) depend only on the electrotonic length L of the cylinder (in units of λ). Specifically, in cylinders of length L with a sealed end, they are given by

$$\tau_0/\tau_i = 1 + (i\pi/L)^2. \quad (5.21)$$

Consequently, Rall showed that L can be estimated directly from the values of τ_i , in particular from the two slowest time constants, τ_0 and τ_1 , that can be “peeled” from the experimental voltage transient (Rall 1969). More details are given in reviews by Rall (1977, 1989) and Jack et al. (1975). You may observe this effect by doing Exercise 5 at the end of the chapter.

Rall also showed that the time course of synaptic potentials changes as the recording point moves away from the input location. The time course of the voltage response near the input site is relatively rapid and it becomes significantly prolonged (and attenuated) at a point distant from the input site. This effect is the source of Rall’s method (Rall 1967) of using shape parameters of the synaptic potentials (its “rise time” and its width at half amplitude, the “half-width”) to estimate the electrotonic distance of the synapse (the input) from the soma (the recording site). As shown in Fig. 5.3B and in Exercise 7, the more delayed the peak is, and the more “smeared” it is, the more distal the input (Rall 1967).

5.4.3 Functional Significance of λ and τ_m

The space constant λ and the membrane time constant τ_m are two very important parameters that play a critical role in determining the spatio-temporal integration of synaptic inputs in dendritic trees. Equation 5.19 shows that τ_m provides a measure of the time window for input integration. A cell with large τ_m (e.g., 50 msec) integrates inputs over a larger time

window compared to cells with smaller τ_m values (say, 5 msec). The value of τ_m depends on the electrical properties of the membrane R_M and C_M , but it does not depend on the cell morphology. Neurons with a high density of open membrane channels (i.e., with a small R_M value) will respond quickly to the input and will “forget” this input rapidly. In contrast, neurons with relatively few open membrane channels (large R_M) will be able to summate inputs for relatively long periods of time (slow voltage decay).

In contrast to τ_m , the space constant λ depends not only on the membrane properties but also on the specific axial resistance and the diameter. In neurons with large λ (e.g., with large R_M and/or large diameter, or small R_A) voltage attenuates less with distance as compared to neurons with a smaller λ value. Thus, in the former case, inputs that are anatomically distant from each other will summate better (spatially) with each other than in the latter case. Therefore, knowledge of λ and τ_m for a given neuron provides important information about the capability of its dendritic tree to integrate inputs both in time and in space.

5.4.4 The Input Resistance R_{in} and “Trees Equivalent to a Cylinder”

A third important parameter is R_{in} , the *input resistance* at a given point in the dendritic tree. When a steady current I_0 is applied at a given location in a structure, a steady voltage V_0 is developed at that point. The ratio V_0/I_0 is the input resistance at that point. This parameter is of great functional significance because it provides a measure for the “responsiveness” of a specific region to its inputs. It is also a quantity which, unlike R_M , may be directly measured. From Ohm’s law, we know that a dendritic region with a large R_{in} requires only a small input current (a small excitatory conductance change) to produce a significant voltage change locally, at the input site. Conversely, a region with small R_{in} requires a more powerful input (or several inputs) to generate a significant voltage change locally.

In the case of an infinite cylinder, when a steady current input is injected at some point $x = 0$, the input current must divide into two equal core currents; one half flows to the right at $x = 0$ and the other half flows to the left. Thus, from Eq. 5.1,

$$I_i = -\frac{1}{r_i} \frac{\partial V}{\partial x} \Big|_{x=0} = \frac{I_0}{2}. \quad (5.22)$$

From Eq. 5.13, the derivative $(\partial V/\partial x)|_{x=0} = -V_0/\lambda$. We then get,

$$R_{in} = V_0/I_0 = r_i \lambda / 2 = \sqrt{r_m r_i} / 2, \quad (5.23)$$

or

$$R_{in} = (1/\pi) d^{-3/2} \sqrt{R_M R_A}. \quad (5.24)$$

For the semi-infinite cylinder, the input resistance (often represented by R_∞) will be twice this amount. Hence, in an infinitely long cylinder, the input resistance is directly

proportional to the square root of the specific membrane and axoplasm resistivities, and is inversely proportional to the core diameter, raised to the 3/2 power. Consequently, thin cylinders have a larger R_{in} compared to thicker cylinders that have the same R_M and R_A values. The dependence of the input resistance on $d^{3/2}$ was utilized by Rall (1959) to develop the concept of “trees equivalent to a cylinder.” Rall argued that when a cylinder with diameter d_p bifurcates into two daughter branches with diameters d_1 and d_2 (and both daughter branches have the same boundary conditions at the same value of L), the branch point behaves as a continuous cable for current that flows from the parent to daughters, if

$$d_p^{3/2} = d_1^{3/2} + d_2^{3/2}. \quad (5.25)$$

Provided that the specific properties of membrane and cytoplasm are uniform, Eq. 5.25 implies that the sum of input conductances of the two daughter branches (at the branch point) is equal to the input conductance of the parent branch at this point (impedance matching at the branch point). Thus, a branch point that obeys Eq. 5.25 is electrically equivalent to a uniform cylinder (looking from the parent into the daughters). Rall extended this concept to trees and showed that (from the soma viewpoint out to the dendrites) there is a subclass of trees that are electrically equivalent to a single cylinder whose diameter is that of the stem (near the soma) dendrite. (See Rall (1959, 1989) and Jack et al. (1975).) It was surprising to find that dendrites of many neuron types (e.g., the α -motoneuron in Fig. 5.1B) obey, to a first approximation, the $d^{3/2}$ rule (Eq. 5.25). (See, for example, Bloomfield, Hamos and Sherman (1987).) However, the dendrites of several major types of neurons (e.g., cortical and hippocampal pyramidal cells) do not obey this rule. Still, the “equivalent cylinder” model for dendritic trees allows for a simple analytical solution (Rall and Rinzel 1973, Rinzel and Rall 1974) and, indeed, it provided the main insights regarding the spread of electrical signals in passive dendritic trees, as summarized in Sec. 5.7.

It may be shown that the input resistance for a finite cylinder with sealed end at $X = L$ is larger by a factor of $\coth(L)$ than that of a semi-infinite cylinder having the same membrane and axial resistance, and the same diameter. When the end at $X = L$ is clamped to rest, the input resistance is smaller than that of the semi-infinite cylinder by a factor of $\tanh(L)$. (See Rall (1989) for complete derivations.) This leads to the useful result that, if the neuron and its associated dendritic tree can be approximated by an equivalent sealed end cylinder of surface area A and electrotonic length L , then

$$R_M = R_{in} A \tanh(L)/L. \quad (5.26)$$

This provides a way to estimate the specific membrane resistance R_M from the measured input resistance if A and L are known, or to estimate the dendritic surface area if R_M and L are known (Rall 1977, 1989).

5.4.5 Summary of Main Results from the Cable Equation

In view of the solutions for the three representative cases, the infinite cylinder (Eq. 5.13), the finite cylinder with sealed end (Eq. 5.14) and the finite cylinder with end clamped to the resting potential (Eq. 5.15), it is important to emphasize a few points:

1. The attenuation of steady voltage is determined solely by the space constant λ , only in the case of infinite cylinders. In this case, steady voltage attenuates e -fold per unit of length λ . In finite cylinders, however, λ is not the sole determinant of this attenuation; the electrical length of the cylinder L and the boundary condition at the end toward which current flows (and voltage attenuates) also determine the degree of attenuation along the cylinder (Fig. 5.2).
2. The “sealed end” or “open circuit” boundary condition and the “clamped to rest” termination are two extremes. In dendritic trees, short dendritic segments terminate by a subtree that imposes “leaky” boundary conditions at the segment’s ends. The size of the subtree and its electrical properties determine how leaky the conditions are at the boundaries of any given segment. In general, when a large tree is connected at the end of the dendritic segment, the leaky boundary condition at this end approaches the “clamped to rest” condition. When current flows in the direction of such a leaky end, voltage attenuates steeply along this segment. In contrast, when the subtree is very small the leaky boundary conditions approach the “sealed-end” condition and a very shallow attenuation is expected towards such an end (Fig. 5.3A). Rall (1959) showed how to compute analytically the various boundary conditions at any point in a passive tree with arbitrary branching and specified R_M , R_A and (for the transient case) C_M values (see also Jack et al. (1975), Segev et al. (1989)).
3. An important consequence of this dependence of voltage attenuation on the boundary conditions in dendritic trees is that this attenuation is asymmetric in the central (from dendrites to soma) vs. the peripheral (away from soma) directions. In general, because the boundary conditions are more “leaky” in the central direction, voltage attenuation in this direction is steeper than in the peripheral one. Figure 5.3A illustrates this important point very clearly.
4. Dendritic trees can be approximated (electrically), to a first degree, by a single (finite) cylinder. Therefore, analysis of the behavior of voltage in such cylinders provides important insights into the behavior of voltage in dendritic trees.
5. By peeling the slowest (τ_0) and the first equalizing time constant (τ_1) from somatic voltage transients, the electrical length L of the dendritic tree could be estimated, assuming that the tree is equivalent to a single cylinder (Eq. 5.21). Indeed, utilizing this peeling method for many neuron types we know that, depending on the neuron

type and experimental condition, L ranges between 0.3–2 (in units of λ). Thus, from the viewpoint of the soma, dendrites are electrically rather compact.

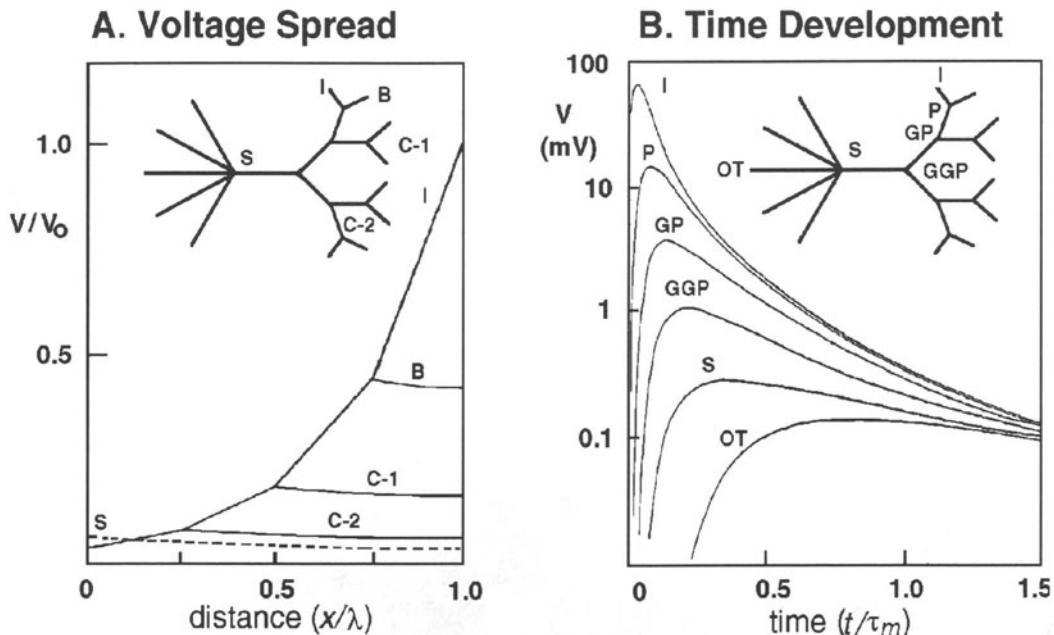


Figure 5.3 The voltage spread in passive dendritic trees is asymmetrical (A); its time-course changes (is broadened) and the peak is delayed as it propagates away from the input site (B). Solid curve in (A) shows the steady-state voltage computed for current input to terminal branch I. Large attenuation is expected in the input branch whereas much smaller attenuation exists in its identical sibling branch B. The dashed line corresponds to the same current when applied to the soma. Note the small difference, at the soma, between the solid curve and the dashed curve, indicating the negligible “cost” of placing this input at the distal branch rather than at the soma. (Data replotted from Rall and Rinzel (1973).) In (B), a brief transient current is applied to terminal branch I and the resultant voltage at the indicated points is shown on a logarithmic scale. Note the marked attenuation of the peak voltage (by several hundredfold) from the input site to the soma and the broadening of the transient as it spreads away from the input site. (Data replotted from Rinzel and Rall (1974).) Dendritic terminals have sealed ends in both (A) and (B).

5.5 Compartmental Modeling Approach

The compartmental modeling approach complements cable theory by overcoming the assumption that the membrane is passive and the input is current (Rall 1964). Mathematically, the compartmental approach is a finite-difference (discrete) approximation to the (nonlinear) cable equation. It replaces the continuous cable equation by a set, or a matrix, of ordinary differential equations and, typically, numerical methods are employed to solve this system

(which can include thousands of compartments and thus thousands of equations) for each time step. Conceptually, in the compartmental model dendritic segments that are electrically short are assumed to be isopotential and are lumped into a single R-C (either passive or active) membrane compartment (Fig. 5.4C). Compartments are connected to each other via a longitudinal resistivity according to the topology of the tree. Hence, differences in physical properties (e.g., diameter, membrane properties, etc.) and differences in potential occur between compartments rather than within them. It can be shown that when the dendritic tree is divided into sufficiently small segments (compartments) the solution of the compartmental model converges to that of the continuous cable model. A compartment can represent a patch of membrane with a variety of voltage-gated (excitable) and synaptic (time-varying) channels. A review of this very popular modeling approach can be found in Segev, Fleshman and Burke (1989).

As an example, consider a section of a uniform cylinder, divided into a number of identical compartments, each of length l . If we introduce an additional current I_j to represent the flow of ions from the j th compartment through active (nonlinear synaptic and/or voltage-gated) channels, we can write Eq. 5.3 as

$$\frac{l^2}{R_a} \frac{\partial^2 V_j}{\partial x^2} = C_m \frac{\partial V_j}{\partial t} + \frac{V_j}{R_m} + I_j. \quad (5.27)$$

Here, V_j represents the voltage in the j th compartment, and we have used Eqs. 5.4-5.6 to give the actual values of the resistances and capacitances (in ohms and farads) of this compartment, instead of the values for a unit length. Note that now R_m represents the membrane resistance at rest, before the membrane potential (and membrane resistance) is changed due to the current I_j . Also, note that V_j appears in the expression for I_j . For example, in the case of synaptic input to compartment j , $I_j = g(t)(V_j - E_{syn})$. Here, $g(t)$ and E_{syn} are synaptic conductance and reversal potential, respectively. (This is discussed in more detail in Sec. 6.3.1 of the following chapter.)

It can be shown by use of Taylor's series that for small values of l , the left hand side of Eq. 5.27 can be expressed in terms of differences between the value of V_j and the values in the adjacent compartments, V_{j-1} and V_{j+1} . In this approximation, the cable equation becomes

$$\frac{V_{j+1} - 2V_j + V_{j-1}}{R_a} = C_m \frac{dV_j}{dt} + \frac{V_j}{R_m} + I_j. \quad (5.28)$$

For the general case of a dendritic cable of non-uniform diameter (in which R_m , R_a and C_m may vary among compartments), we obtain the result given earlier in Eq. 2.1 and discussed in Sec. 2.2. This equation can easily be extended to include a branch structure. For a tree represented by N compartments we get N coupled equations of the form of Eq. 5.28. They should be solved simultaneously to obtain V_j , for $j = 1, 2, \dots, N$ at each time step Δt .

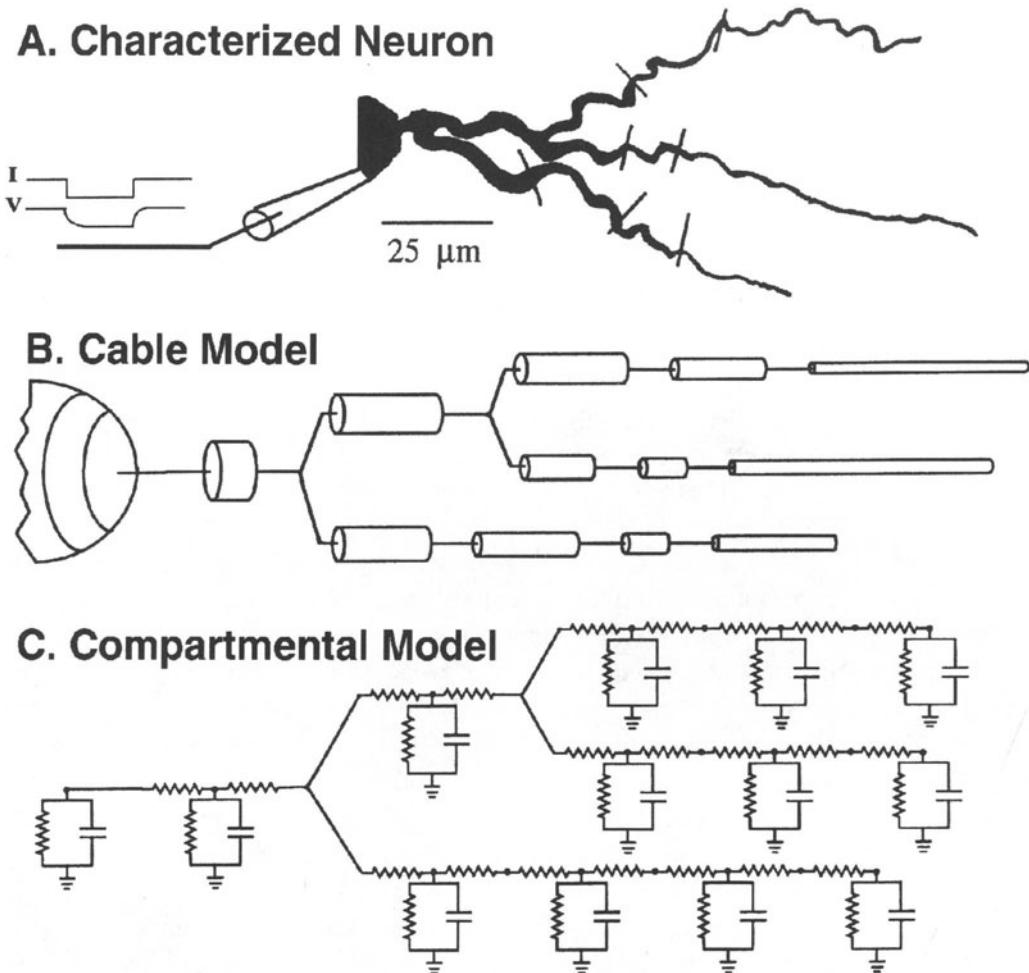


Figure 5.4 Dendrites (A) are modeled either as a set of cylindrical membrane cables (B) or as a set of discrete isopotential R-C compartments (C). In the cable representation (B), the voltage can be computed at any point in the tree by using the continuous cable equation and the appropriate boundary conditions imposed by the tree. An analytical solution can be obtained for any current input in passive trees of arbitrary complexity with known dimensions and known specific membrane resistance and capacitance (R_M , C_M) and specific cytoplasm (axial) resistance (R_A). In the compartmental representation, the tree is discretized into a set of interconnected R-C compartments. Each is a lumped representation of the membrane properties of a sufficiently small dendritic segment. Compartments are connected via axial cytoplasmic resistances. In this approach, the voltage can be computed at each compartment for any (nonlinear) input and for voltage and time-dependent membrane properties.

A discussion of the numerical techniques used by GENESIS to solve this set of equations is given in Chapter 20.

5.6 Compartmental Modeling Experiments

Having covered the basics of cable theory and compartmental modeling, we are now ready to try some computer “experiments” that will help us to better understand the previous sections. The GENESIS *Cable* tutorial implements a dendritic cable model that has been converted to an equivalent cylinder. Thus, it creates a one-dimensional compartmentalized cable similar to a single branch of the dendritic tree shown in Fig. 5.4C. Each cylindrical compartment is similar to the one shown in Fig. 2.3, with the axial resistance on the left side of the compartment and with the resting membrane potential E_m set to zero. You can provide a current injection pulse to any compartment, setting the delay, pulse width and amplitude from a menu. Although we won’t make use of it in this chapter, you may also provide a synaptic input, corresponding to the variable resistor in series with a battery shown in Fig. 2.3.

The cable consists of a “soma” compartment and N identical “cable” compartments. You can set N to be any number you like, and can separately modify the length and diameter for the soma and for the identical cable compartments. You can also set the specific axial and membrane resistances and specific membrane capacitance. These values apply to both the soma and cable compartments. The soma compartment is labeled as compartment 0, and is the leftmost compartment, located at $X = 0$. This means that the axial resistance of the soma compartment doesn’t enter into the calculation — all compartments communicate through the axial resistances of the dendritic cable. This also means that the rightmost compartment, compartment N at $X = L$, can have no current flow to the right. Thus, our model corresponds to the sealed end boundary condition discussed in Sec. 5.4.1.

As always, the best way to understand the model is to run the simulation. Before reading further, you should start the *Cable* simulation from a terminal window at the bottom of your screen. This is done by changing to the *Scripts/cable* directory and typing “genesis Cable.” After a slight delay, a control panel and two graphs will appear. If you click the mouse on both the Change Cell Parameters and Change Current Injection buttons, two more windows should appear, resulting in a display similar to the one shown in Fig. 5.5.

As with most of the tutorials, the HELP button will call up a menu with a number of selections, including Running the Simulation. You may use this to get a description of the uses of the various buttons, toggles and dialog boxes. Take a few minutes to understand what the default values of the cable parameters are, and the units that are being used. Note that the *Cable Compts.* dialog box shows “0,” indicating that there is only a single soma compartment. The dialog boxes in the Cell Parameters menu at the upper right indicate

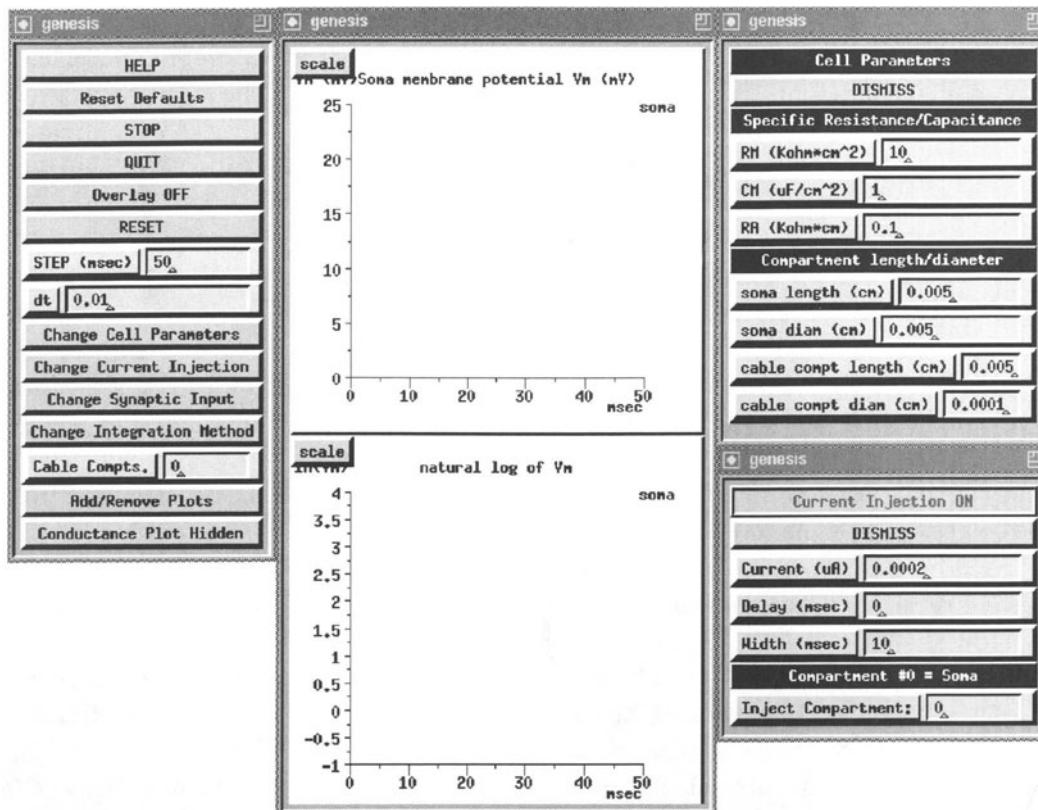


Figure 5.5 A display from the GENESIS *Cable* tutorial, showing the default values of various parameters. The Change Cell Parameters and Change Current Injection buttons on the control panel (to the left) were used to call up the two menus on the right.

that it has a length and diameter both equal to $50 \mu m$ ($0.005 cm$). Later, we will add dendritic cable compartments and change some of the parameters.

Try hitting “Return” while the cursor is in the data field of one of the dialog boxes in the Cell Parameters menu. This would normally be done after changing some of the dialog boxes for the compartment dimensions or the specific resistances and capacitances, so that the values of R_m , C_m and R_a will be recomputed. It also causes these values to be displayed in the terminal window. Can you satisfy yourself that they are the ones expected from Eqs. 5.4–5.6? The menu at the lower right indicates that, with no delay, a current of $0.2 nA$ will be injected to the soma for $10 msec$. Can you predict what the plots of $V(t)$ and $\ln(V(t))$ will look like? What will be the maximum value of $V(t)$? What do you expect the slope of the logarithmic plot to be after the end of the injection pulse? After you have made these estimates, go ahead and click on STEP to perform the simulation for

50 msec. Now, measure the quantities that you have predicted. If the values are not within reasonable agreement with theory, take the time to correct your mistakes, or to understand what went wrong. (Hint: use the scale button to call up the graph scale menu, or click and drag the mouse along a graph axis to zoom in to a region in order to make more accurate measurements. It may be helpful to use a small ruler with a millimeter scale to help with interpolation. If your computer has the ability to capture and print portions of the screen image, this will make these measurements even easier.)

After this “warm-up” exercise, we are ready to perform a more interesting experiment. Now, add a dendritic cable to the soma by changing the **Cable Compts.** dialog box contents to “10.” Keep the original dimensions for the soma compartment and also use the default values of the cable compartment dimensions (a diameter of 1 μm and length of 50 μm). You should be able to verify that each cable compartment has a length of 0.1λ , and that the electrotonic length of the entire cable is given by $L = 1$. Although we are not dealing with a steady voltage, nor with an infinite cable, Eq. 5.13 suggests that the change in voltage from one compartment to another will be relatively small with this size compartment, justifying our use of a compartmental model. Clearly, the question of how small we need to make the compartments is an important one. You may investigate this issue more carefully in Exercise 3, at the end of this chapter.

Before continuing, you should be aware of another factor that can affect the accuracy of your simulation. In previous chapters, we have referred to the importance of choosing an appropriate size for the integration time step (given in the **dt** dialog box). This is particularly true for cable models that contain many short compartments (Chapter 20, Sec. 20.3.4). Although the default time step should be adequate for the exercises suggested here, you should experiment with smaller values if you make significant changes in the cable parameters.

In order to compare the results to those obtained with only the isolated soma, click on **Overlay OFF** so that it changes to **Overlay ON**; then click on **RESET** and **STEP**. Do you expect the different dimensions of the cable to affect the decay of the voltage after the pulse? Can you explain any differences in the two plots of $\ln(V)$ right after the end of the injection pulse? You may explore this effect in more detail in Exercise 5.

Finally, let’s examine the effect of the dendrite diameter on the propagation of voltage changes along the dendritic cable. For this, we would like a little sharper change in voltage, so increase the soma injection current to 2 nA (0.002 μA) and reduce the pulse width to 1 msec. In order to measure the potential at other places along the cable, click on the **Add/Remove Plots** button to bring up a menu that will allow you to specify other compartments in which to record the membrane potential. Note that the default is to record from the soma. By entering “5” and then “10” in the dialog box, you may add plots of V for these two compartments. Clicking on **Remove Cable Plots** removes plotting for all but the soma compartment. Having added these two extra plots, leave the cable diameter at 1 μm and run the simulation. Make a note of the maximum value of V_m and the time it takes to reach this

value for each of the three locations. Now, change the cable diameter to $0.5 \mu\text{m}$ and repeat the experiment. What is the electronic length L of the cable with this new diameter? How does the diameter affect the decay of the potential with distance? Can you explain this? By observing the change of position of the peak in V with distance, can you estimate the propagation velocity of the voltage deviation? How does the propagation velocity seem to depend on dendrite diameter?

There are a number of other experiments suggested at the end of the chapter that will shed more light on the effect the dendritic cable has upon the propagation of electrical signals. For example, Exercise 4 investigates the asymmetry of propagation from soma to dendrites as compared to propagation in the opposite direction, as seen in Fig. 5.3A. Exercise 6 studies the effect of the cable on the input resistance of the neuron. In Exercise 7, we make use of the broadening of potentials as they propagate away from the point at which they were introduced (Fig. 5.3B) to estimate dendritic lengths.

5.7 Main Insights for Passive Dendrites with Synapses

Here we summarize the main insights regarding the input-output properties of passive dendrites, gained from modeling and experimental studies on dendrites during the last forty years.

1. Dendritic trees are electrically distributed (rather than isopotential) elements. Consequently, voltage gradients exist over the tree when synaptic inputs are applied locally. Because of inherent non-symmetric boundary conditions in dendritic trees, voltage attenuates much more severely in the dendrites-to-soma direction than in the opposite direction (Fig. 5.3A). In other words, from the *soma* viewpoint, dendrites are electrically rather compact (average cable length L of $0.3\text{--}2 \lambda$). From the *dendrite* (synaptic) viewpoint, however, the tree is electrically far from being compact. A corollary of this asymmetry is that, as seen in Fig. 5.3A, short side branches (in particular, dendritic spines) are essentially isopotential with the parent dendrite when current flows from the parent into these side branches (spines) but a large voltage attenuation exists from terminal (spine) to parent when the terminal receives direct input.
2. The large voltage attenuation from dendrites to soma (which may be a few hundredfold for brief synaptic inputs at distal sites) implies that many (several tens) of excitatory inputs should be activated within the integration time window τ_m , in order to build up depolarization of $10\text{--}20 \text{ mV}$ and reach threshold for firing of spikes at the soma and axon (e.g., Otmakhov, Shirke and Malinow 1993, Barbour 1993). The large local depolarization expected at the dendrites, together with the marked attenuation in the

tree imply that the tree can be functionally separated into many, almost independent, subunits (Koch, Poggio and Torre 1982).

3. Although severely attenuated in peak values, the attenuation of the area of transient potentials (as well as the attenuation of charge) is relatively small. The attenuation of the area is identical to the attenuation of the steady voltage, independently of the transient shape (Rinzel and Rall 1974). Comparing the steady-state somatic voltage that results from distal dendritic input to the soma voltage when the same input is applied directly to the soma (dashed line in Fig. 5.3A) highlights this point. Thus, the “cost” (in terms of area or charge) of placing the synapse at the dendrites rather than at the soma is quite small.
4. Linear system theory implies an interesting reciprocity in passive trees. The voltage at some location x_j resulting from transient current input at point x_i is identical to the voltage transient measured at x_i when the same current input is applied at x_j (Koch et al. 1982). Because the input resistance is typically larger at thin distal dendrites and in spines than at proximal dendrites (and the soma), the same current produces a larger voltage response at distal dendritic arbors. Thus, the reciprocity theorem implies that the *attenuation* of voltage from these sites to the soma is steeper than in the opposite direction (i.e., asymmetrical attenuation). Reciprocity also holds for the total signal delay between any given points in the dendritic tree (Agmon-Snir and Segev 1993).
5. Synaptic potentials are delayed, and they become significantly broader, as they spread away from the input site (Fig. 5.3B). The large sink provided by the tree at distal arbors implies that, locally, synaptic potentials are very brief. At the soma level, however, the time-course of the synaptic potentials is primarily governed by τ_m . This change in width of the synaptic potential implies multiple time windows for synaptic integration in the tree (Agmon-Snir and Segev 1993).
6. Excluding very distal inputs, the cost (in terms of delay) that results from dendritic propagation time (i.e., the net dendritic delay) is small compared to the relevant time window (τ_m) for somatic integration. Thus, for the majority of synapses, the significant time window for *somatic* integration remains τ_m (Agmon-Snir and Segev 1993).
7. Because of the inherent conductance change (shunt) associated with synaptic inputs, synaptic potentials sum nonlinearly (less than linear) with each other (Chapter 6). This local conductance change of the membrane is better “felt” by electrically adjacent synapses than by more remote (electrically decoupled) synapses. Consequently, in passive trees, spatially distributed excitatory inputs sum more linearly (produce more charge) than do spatially clustered synapses (Rall 1964).

8. Inhibitory synapses (whose conductance change is associated with a battery near the resting potential) are more effective when located on the path between the excitatory input and the “target” point (soma) than when placed distal to the excitatory input. Thus, when strategically placed, inhibitory inputs can specifically veto parts of the dendritic tree and not others (Rall 1964, Koch et al. 1982, Jack et al. 1975).
9. Because of dendritic delay, the somatic depolarization that results from activation of excitatory inputs at the dendrites is very sensitive to the temporal sequence of the synaptic activation. It is largest when the synaptic activation starts at distal dendritic sites and progresses proximally. Activation of the same synapses in the reverse order in time will produce smaller somatic depolarization. Thus, the output of neurons with dendrites is inherently directionally selective (Rall 1964).
10. Because the synaptic input changes the membrane conductance, it effectively alters the cable properties (electrotonic length, input resistance, time constant, etc.) of the postsynaptic cell. This activity can reduce the time constant by a factor of 10 (Bernander et al. 1991, Rapp et al. 1992). Thus, spontaneous (background) synaptic activity dynamically changes the computational (input-output) capabilities of the neuron.

5.8 Biophysics of Excitable Dendrites

A growing body of experimental evidence in recent years has clearly demonstrated that the membrane of many types of dendrites is endowed with voltage-gated (nonlinear) ion channels, including the NMDA channels as well as voltage-activated inward (Ca^{+2} and Na^{+}) and outward (K^{+}) conductances (e.g., Stuart and Sakmann 1994, Laurent 1993, McKenna et al. 1992, Wilson 1992). These channels are responsible for a variety of subthreshold electrical nonlinearities and, under favorable conditions, they can generate full-blown action potentials. The use of voltage- and ion-dependent dyes as well as intracellular and patch-clamp recordings from dendrites suggested that, in contrast to axonal trees, the regenerative phenomenon from input into excitable dendrites tends to spread only locally. This makes functional sense since, otherwise, the dendritic tree would be essentially no different from the axon, implementing a simple all-or-none operation. However, because of the asymmetric spread of voltages within the dendritic tree (Fig. 5.3A) and because of inhomogeneous distribution of excitable channels in dendrites, spikes can propagate more readily back from the soma to the dendrites (Stuart and Sakmann 1994). Unfortunately, we still lack information regarding the distribution, the voltage-dependence, and the kinetics of excitable channels in dendrites and most of the results of this section are primarily based on theoretical predictions.

What is the electrical behavior to be expected from dendrites with voltage-gated membrane ion channels? First, we note that the presence of voltage-gated channels in dendrites

does not automatically imply that these channels participate in the electrical activity of the tree under all conditions. The large conductance load imposed by the tree effectively increases the activation threshold of these channels for local synaptic potentials (Rall and Segev 1987). These channels will be more readily activated under favorable conditions such as in regions with high densities of excitable channels (as in the initial segment of the axon), when the excitable channels have fast activation kinetics, or when the input is distributed (not localized). When activated, these channels can modulate the input-output properties of the neuron. For example, they can *amplify* the excitatory synaptic current and, for channels that carry inward current, the regenerative activity can spread (“chain reaction”) and indirectly activate nearby dendritic regions that will further enhance the excitatory synaptic inputs (Rall and Segev 1987). Consequently, distal excitatory synaptic inputs may be less attenuated and, thus, affect more strongly the neuron output than would be expected in the passive case. In general, because of the asymmetry of voltage attenuation in dendritic trees (Fig. 5.3A), regenerative activity in dendrites will spread more securely in the centrifugal (soma-to-dendrites) direction than in the centripetal direction. This effect may be explored in the simulation (*traub91*) of the CA3 pyramidal cell described in Chapter 7. In addition to modulating the strength of the synaptic current, the kinetics of excitable channels may also play an important role in modulating the *speed* of electrical interaction in the dendritic tree.

Another consequence of dendritic nonlinearity was discussed by Mel (1993). Unlike the passive case where synaptic saturation implies loss in synaptic efficacy when the synapses are spatially clustered (see number 7 above), in the excitable situation (including the case of the voltage- and transmitter-gated NMDA receptors) a certain degree of input clustering implies more charge transfer to the soma (due to the extra active inward current). In this case, the output at the axon depends sensitively on the size (and site) of the “clusteron” and this may serve as a mechanism for implementing a multi-dimensional discrimination task of input patterns via multiplication-like operation.

Recently, the possibility that active dendritic currents (both inward and outward) may serve as a mechanism for synaptic gain control was put forward by Wilson (1992) in the context of neostriatal neurons and by Laurent (1993) for the axonless nonspiking interneurons of the locust. The principal idea is that, as a result of active currents, the integrative capabilities of the neuron (e.g., its input resistance and electrotonic length) are dynamically controlled by the membrane potential; thereby the neuron output depends on its state (membrane potential). Active currents (e.g., outward K⁺ current) can act to counterbalance excitatory synaptic inputs (negative feedback) and thus stabilize the input-output characteristics of the neuron. Conversely, at other voltage regimes, active currents might effectively increase the input resistance and reduce the electrotonic distance between synapses (positive feedback) with the consequence of nonlinearly boosting a specific group of coactive excitatory synapses.

5.9 Computational Function of Dendrites

It seems appropriate to conclude this chapter by asking what kind of computations could be performed by a neuron with dendrites that could not be carried out with just a formless point neuron. Several answers have already been discussed; here the major ones are succinctly highlighted:

1. Neurons with dendrites can compute the direction of motion (Rall 1964, Koch et al. 1982).
2. Neurons with dendrites can simultaneously function on multiple time windows. For local dendritic computations (e.g., triggering local dendritic spikes, triggering local plastic processes) distal arbors act more as coincidence detectors, whereas the soma acts more as an integrator when brief synaptic inputs (i.e., non-NMDA and GABA_A) are involved (Agmon-Snir and Segev 1993).
3. Neurons with dendrites can implement a multi-dimensional classification task (Mei 1993).
4. Neurons with dendrites can function as many, almost independent, functional subunits. Each unit can implement a rich repertoire of logical operations (Koch et al. 1982, Rall and Segev 1987) as well as other local computations (e.g., local synaptic plasticity) and they can function as semi-autonomous input-output elements (e.g., via dendrodendritic synapses).
5. Neurons with slow ion currents in the dendrites that are partially decoupled from fast spike-generating currents at the soma/axon hillock can produce a large repertoire of frequency patterns. By modulating the degree of electrical coupling between the dendrites and the soma (e.g., by inhibition) the same input can produce regular high frequency spiking as well as bursting — as thought to occur in experimental and theoretical models of epileptic seizures (Pinsky and Rinzel 1994).

5.10 Exercises

1. Plot the theoretical attenuation of steady voltage as a function of physical distance in infinitely long cylindrical cables with $R_M = 10,000 \Omega \cdot cm^2$ and $R_A = 100 \Omega \cdot cm$. Make plots for the three cable diameters, $d = 0.5 \mu m$, $1 \mu m$ and $2 \mu m$. What can you conclude about the effect of dendritic diameter on the passive propagation of voltages? How does this compare with the results of the experiment on the finite cable discussed in Sec. 5.6?

2. (a) Calculate the input resistance of infinite cylindrical cables with $d = 0.5 \mu m$, $1 \mu m$ and $2 \mu m$. As in the preceding exercise, assume $R_M = 10,000 \Omega \cdot cm^2$ and $R_A = 100 \Omega \cdot cm$.
 - (b) Calculate the input resistance of two identical daughter branches of the above cylinder that obey Eq. 5.25.
 - (c) Calculate the input resistance at $X = 0$ of a finite cylinder with $d = 1 \mu m$, $R_M = 10,000 \Omega \cdot cm^2$ and $R_A = 100 \Omega \cdot cm$. Examine both the case of a sealed end at $X = 1$ and the case of an end that is clamped to rest ($V = 0$) at $X = 1$.
 - (d) Which of the input resistances calculated in (a) and (c) is closest to the result for the finite cable that we simulated in Sec. 5.6? Explain any similarities and differences.
3. One of the first questions to be answered before using or constructing a compartmental model is, “What value of l is small enough to allow the approximation of Eq. 5.27 with Eq. 5.28?” This question can be answered by performing “computer experiments” on a system for which we know the exact solution. Equation 5.14 gives the exact result for the attenuation of a steady-state membrane potential with distance for a uniform finite length cable. In order to make the simulated cable a uniform one, change the soma dimensions to make it the same size as a dendritic compartment. Then add 10 cable compartments to the soma, using the default values of all the parameters. Explain why, although there are now 11 compartments, including the soma, the electrotonic length of this cable is $L = 1$, and not $L = 1.1$.

Set the value of the current injection to the soma at $0.0001 \mu A$ ($0.1 nA$), and set the width of the injection pulse to a large value, so that there will be a constant injection current. Plot the membrane potential in both the soma and the end compartment, and calculate the steady-state ratio of $V(L)/V(0)$. How does it compare with the prediction of Eq. 5.14?

Now, repeat the experiment with 5 and then 20 cable compartments, changing the compartment length to keep the total electrotonic length of the cable at $L = 1$. What do you conclude is a good practical definition of “small enough”?

4. Restore the default parameters for the *Cable* simulation (soma length and diameter of $50 \mu m$; cable compartment diameter of $1 \mu m$ and length of $50 \mu m$). Make a cable with 10 dendritic compartments, and provide a $2 msec$ pulse of $0.1 nA$ current injection to the soma, recording the membrane potential at both the soma and compartment 10. Then change the injection point to compartment 10, toggle to `overlay ON`, click on `RESET` and run the simulation again. Make an estimate of the attenuation of the voltage in the soma-to-dendrite direction and then in the opposite direction. Finally, repeat the experiment with the dendritic cable diameter set to $0.5 \mu m$. What accounts for the

different attenuation in the two directions? Why is the dendritic diameter relevant? What significance do these results have for neuronal behavior?

5. Use the uniform cable ($L = 1$) from Exercise 3 and apply a brief (2 msec) current injection to the soma. Use the Add/Remove Plots menu to plot $V(t)$ from compartments 0, 5 and 10. Examine the plot of $\ln(V)$ vs. t and note that the response is not a simple exponential. Explain why the three logarithmic plots have different curvatures at the start of the voltage decay, and account for the direction of each.

Now, record only from the soma, and generate overlaid logarithmic plots for cables of length $L = 0.5, 1$ and 2 (5, 10 and 20 compartments). Explain why some of these show a linear slope at earlier times than others.

6. Use the *Cable* simulation to construct a uniform cable like the one in Exercise 3 and perform an experiment to measure the input resistance at the soma ($X = 0$). Use this value to calculate the specific membrane resistance R_M for the cable and compare it to the value that was actually used in the simulation.
7. Using the *Cable* simulation, add 15 cable compartments, each 0.1λ long, to the soma compartment. Place your recording electrode at compartment 0 (“soma”) and inject a brief current pulse once at compartment 0, then at 5, 10 and 15. Measure the peak time (PT) and half width (HW) of the somatic voltage for the different cases and plot PT vs. HW for the different X values. Label each data point with the X value to which it corresponds. Do some interpolation. Can you predict the result for compartment 8 with any accuracy?

Chapter 6

Temporal Interactions Between Postsynaptic Potentials

IDAN SEGEV

6.1 Introduction

The previous two chapters have introduced two of the essential ingredients for the description of neuronal behavior. Chapter 5 has discussed the passive propagation of synaptic inputs through the dendritic tree to the soma and the initial axon segment. Here, voltage-activated channels (Chapter 4) respond to produce the action potentials that are conducted along the axon, resulting in a release of neurotransmitters at the presynaptic terminals. The present chapter deals with the response of the postsynaptic region to this input — namely, with the development of the *postsynaptic potential* (PSP).

Synaptic inputs from different presynaptic sources converge onto the postsynaptic neuron; typically onto its soma-dendritic membrane surface. There these inputs interact with each other and are integrated before output is produced in the axon. The number of synaptic inputs, their characteristics, as well as their spatial and temporal distribution vary in different cell types and in the same cell under various conditions. Some neurons receive only a few synaptic inputs whereas a typical neuron in the mammalian central nervous system (CNS) may receive several thousands of such inputs. Nevertheless, the principles that govern the interactions among postsynaptic potentials both in time and in space is relatively well understood. In this chapter, we focus on the temporal aspect of this interaction.

Here we treat the events that take place locally, at the postsynaptic membrane. We

analyze the initiation of the postsynaptic potential following the opening of synaptic channels (i.e., a conductance change) induced by the release of the neurotransmitter from the presynaptic terminal. First, the basic (R-C) electrical model of a neuronal membrane is introduced. Then a synaptic branch is added to this analog circuit and the production of the PSP is discussed. The case of several inputs impinging on the same patch of membrane at different times is also considered. Finally, we use the GENESIS *Neuron* tutorial together with several suggested exercises in order to gain a better understanding of the significance of the temporal interaction between several excitatory and/or inhibitory synapses for the input-output function of neurons.

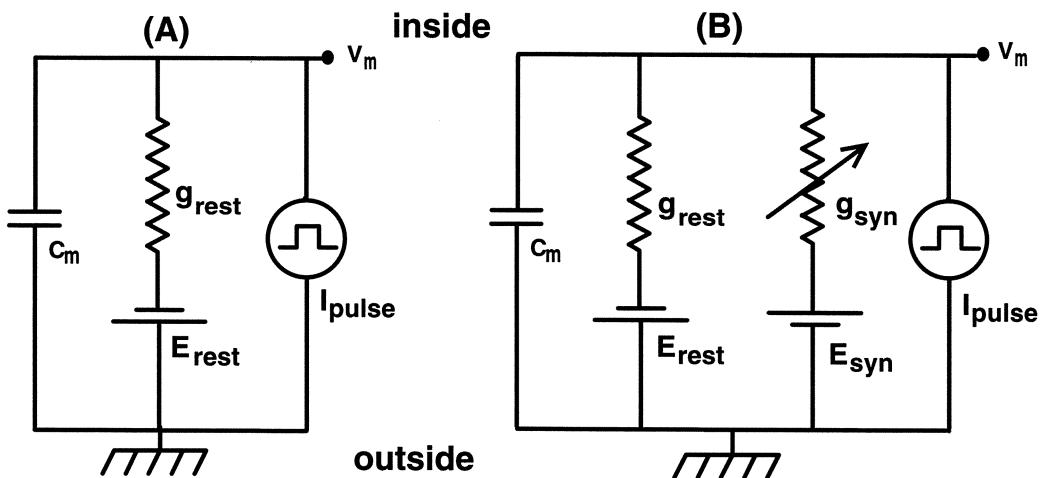


Figure 6.1 Equivalent circuits for electrical model of an isopotential nerve membrane. In (A), the passive (voltage- and time-independent) membrane elements are shown. The resting conductance g_{rest} represents the total conductance of all the transmembrane ion channels that are opened at the resting potential E_{rest} . The capacitance C_m represents the non-channel part of this membrane, and V_m is the voltage difference between the cell interior and the cell exterior. In (B), an additional conductive branch is added in parallel with the passive elements to represent the synaptic channels in the membrane. The total time-dependent conductance change induced by the activation of the synapse is denoted by g_{syn} ; the associated reversal potential (the synaptic battery) is E_{syn} . I_{pulse} represents an externally applied current injection pulse.

6.2 Electrical Model of a Patch of Membrane

Figure 6.1 depicts the electrical circuit for a small isopotential patch of membrane that consists of two types of transmembrane channel. The passive channels are modeled by a constant (time- and voltage-independent) conductance (g_{rest} , in siemens = $1/\Omega$) in series with a fixed voltage source (E_{rest} , in volts) that designates the resting potential. The synaptic (chemically gated) channels are modeled by a separate conductive branch consisting of

a time-dependent conductance $g_{syn}(t)$ in series with a constant voltage source E_{syn} , the reversal potential of the synaptic current. In parallel with these conductive branches there is the capacitive branch C_m which models the dielectric properties of the lipid bilayer. Notice the correspondence between this figure and the “generic” neural compartment discussed in Chapter 2 and shown in Fig. 2.3. From this circuit, it is clear that the voltage difference V_m across the membrane depends on the values of the conductances and batteries involved in the circuit; V_m is expected to change when the values of the conductances g_{rest} and g_{syn} change. This is exactly what happens when a synapse is activated and the transmitter-gated channels are opened at the postsynaptic membrane. To clarify the principles that govern the voltage changes in the circuit of Fig. 6.1B, let us start with the simple case in which a constant current I_{pulse} is injected through an electrode across a patch of a membrane that consists of only passive channels. The corresponding model is depicted in Fig. 6.1A.

6.2.1 Voltage Response of Passive Membrane to a Current Pulse

When the membrane patch does not contain synaptic channels (or when all the synaptic channels are closed, i.e., $g_{syn} = 0$), the circuit in Fig. 6.1B collapses into a simple R-C circuit (Fig. 6.1A). The value of g_{rest} is then the input conductance (1/input resistance) of this membrane patch (hereafter designated as the “compartment”). As previously noted, E_{rest} is the resting potential of this compartment (it is the reversal potential of the ions that flow through the passive channels). The value of E_{rest} is described by the classical equation of Goldman (see, for example, Jack et al. 1975). The Goldman equation shows that when different ion species (e.g., K^+ , Na^+ , Ca^{+2} , Cl^-) flow through transmembrane channels, the equilibrium potential (the resting potential) is determined by the relative permeability to the different ions and by their gradients across the membrane. As seen by the polarity of E_{rest} , the resting potential in nerve cells is negative; namely, the interior of nerve cells is more negative than the exterior.

The membrane capacitance C_m represents the effect of the lipids of the membrane; they are poor conductors and are able to store charges (ions) on either side of the membrane. The membrane resistance (also, in the isopotential case, the input resistance) is $R_m = 1/g_{rest}$.

When no current is injected into the compartment (and, therefore, the net current across the membrane is zero), the membrane voltage V_m remains at the resting potential. However, when a current pulse I_{pulse} is injected between the two sides of the membrane, the voltage across the membrane changes. The charging of the membrane capacitance by injection currents may be represented by an equation analogous to Eq. 4.1, which we used for the Hodgkin–Huxley model. According to Kirchoff’s current law, the net current leaving the compartment (the algebraic sum of the capacitive current flow $I_C = C_m dV_m/dt$ which charges the membrane capacitance and the ionic current I_{rest} that flows across the membrane through the resting channels) should equal the injected current I_{pulse} . Hence,

$$I_C + I_{rest} = I_{pulse}. \quad (6.1)$$

Note that, as we have done in Chapters 4 and 5, we are using the “physiologists’ convention” in which the ionic channel current (I_{rest}) is considered to be positive when positive charge flows *out* of the compartment. For this reason, I_{rest} and the inward current I_{pulse} appear on opposite sides of the equation. By analogy with Eq. 4.3, the ionic current at rest I_{rest} is $g_{rest}(V_m - E_{rest})$. Equation 6.1 then becomes

$$C_m \frac{dV_m}{dt} + g_{rest}(V_m - E_{rest}) = I_{pulse}. \quad (6.2)$$

For simplicity, from now on we will set E_{rest} to zero, as we have done in the previous two chapters. This way, all voltage displacements and batteries are measured with respect to the resting potential. Equation 6.2 is now

$$C_m \frac{dV_m}{dt} + g_{rest} V_m = I_{pulse}. \quad (6.3)$$

If we assume that the current injection pulse begins at time $t = 0$, when $V_m(0) = 0$, we may solve Eq. 6.3 by separating variables and obtain:

$$V_m(t) = \frac{I_{pulse}}{g_{rest}} (1 - e^{-g_{rest}t/C_m}) \quad (6.4)$$

or:

$$V_m(t) = I_{pulse} R_m (1 - e^{-t/\tau_m}). \quad (6.5)$$

In Eq. 6.5, we have defined the *membrane time constant* to be $\tau_m = C_m/g_{rest} = R_m C_m$. For an isolated isopotential compartment, the input resistance (Sec. 5.4.4) is equal to the membrane resistance R_m . As noted in Chapter 5, this is given by the specific resistance of the membrane R_M divided by the surface area A of the compartment. Because nerve cells vary in size and in the specific properties of the membrane, the value of R_m also varies in different cell types and may range from less than $1 M\Omega$ to several hundred $M\Omega$ ($M = 10^6$). However, from the definitions of C_M and R_M (Eqs. 5.4 and 5.5), we can write $\tau_m = R_m C_m = R_M C_M$ (Eq. 5.8). Thus, although the input resistance and membrane capacitance individually vary with the cell area, the time constant is independent of area. Hence, the time constant of nerve cells does not depend on the dimensions of the cell but only on the properties of its membrane. As also mentioned in Chapter 5, C_M is close to $1 \mu F/cm^2$ in most biological membranes. However, as the value of R_M varies among different cell types (the density of channels that are opened at rest varies) the time constant of these cells is different and may range from under $1 msec$ to several hundred milliseconds.

When calculating values of τ_m , it is important to be consistent in one’s choice of units. In the SI (MKS) system of units, resistance and capacitance are measured, respectively, in

ohms and farads, and $R_m C_m$ has units of seconds. However, it is often more convenient to use “physiological units” in which resistance is measured in $K\Omega$ (kilohms) and capacitance is measured in μF . The corresponding units for R_M and C_M would then be $K\Omega \cdot cm^2$ and $\mu F/cm^2$, meaning that τ_m will be expressed in milliseconds. We use these units in this chapter, and in the *Neuron* tutorial. The relationship between SI and physiological units is discussed in more detail in Chapter 13, and summarized in Table 13.1. As discussed below and in Chapter 5, the time constant and the input resistance have important consequences for the electrical behavior of nerve cells and many electrophysiological experiments are aimed at estimating their values for the cell under study.

Observing Eq. 6.5, one sees that during the application of a positive current pulse to the interior of the cell the membrane potential increases (depolarizes) exponentially from the resting potential (0) toward the maximal (steady-state) value $I_{pulse} R_m$. This is shown in Fig. 6.2. The rise is governed by the single time constant τ_m , which is equal to the time at which the voltage rises to 63% ($1 - e^{-1}$) of its maximal (steady-state) value. The steady-state value is reached when the current is injected for an infinitely long duration (i.e., when $t \rightarrow \infty$). Then, the capacitive (time-dependent) current is zero and the membrane current is solely a resistive (ohmic) current. In the other extreme, when the duration of the pulse is very short, most of the injected current flows through the capacitance. In this case, the voltage response is almost independent of the input resistance of the cell, since very little current flows through g_{rest} . The latter statement can be proven by expanding Eq. 6.5 in a Taylor’s series near $t = 0$, neglecting nonlinear terms in t .

Until now, we have analyzed the development of membrane voltage while the current injection is still on. Suppose that the current pulse lasts for a duration of $t = t_{pulse}$; what happens at $t > t_{pulse}$? Now the external current source terminates and, therefore, the net current through the membrane is zero. In this case, the charge on the membrane capacitance dissipates by flowing through the resting ionic channels and

$$C_m \frac{dV_m}{dt} + g_{rest} V_m = 0; t \geq t_{pulse}. \quad (6.6)$$

The solution is

$$V_m(t) = V_m(t_{pulse}) e^{-(t-t_{pulse})/\tau_m}; t \geq t_{pulse}. \quad (6.7)$$

Thus, following the termination of the current pulse, the voltage decays (repolarizes) exponentially from the maximal value $V_m(t_{pulse})$, obtained at the end of the rectangular current pulse, toward the resting level. It decays with the same time constant (τ_m) as it rises during the current pulse. If you wish, you may use the *Cable* tutorial from Chapter 5, or the *Neuron* tutorial which we use in this chapter to verify that the results are as shown in Fig. 6.2.

Before leaving this section, let us briefly mention the implications of τ_m and R_m for the integrative capabilities of nerve cells. As demonstrated in Fig. 6.2, the time constant (Eqs. 6.5

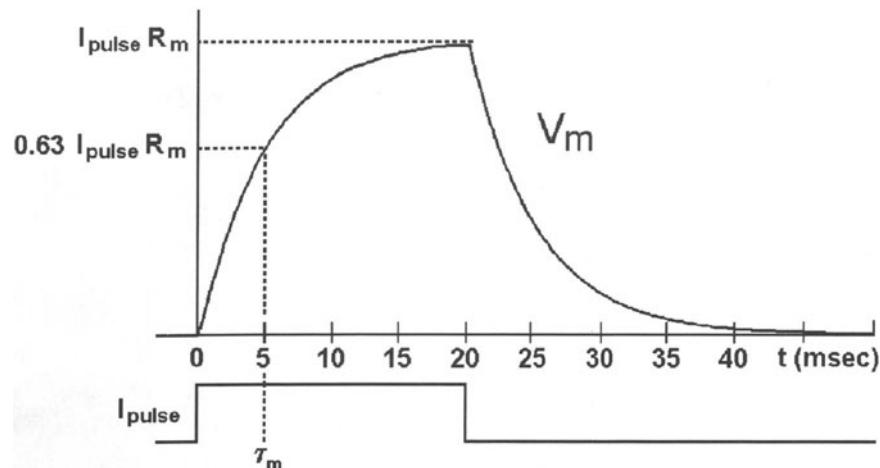


Figure 6.2 The response of the passive circuit in Fig. 6.1 to a positive rectangular current pulse. The current I_{pulse} is given for a duration of 20 msec (bottom trace) and the resultant voltage V_m across the membrane is shown above. The input resistance of the cell is denoted by R_m ($= 1/g_{rest}$). The time constant ($\tau_m = R_m C_m$) of this cell was set to 5 msec. Hence, V_m approaches the maximal (steady-state) value $I_{pulse} R_m$. As expected from Eq. 6.5, V_m reaches $(1 - 1/e)$, or 63% of its maximal value at $t = \tau_m$. This time constant also governs the exponential decay of V_m at the end of the current application. Then, $V_m = V_0 e^{-(t-t_{pulse})/\tau_m}$, where V_0 is the maximal voltage reached at $t = t_{pulse} = 20$ msec.

and 6.8) implies that the buildup of voltage at the postsynaptic membrane as a response to an input takes time, and that when the input ends the membrane “remembers” the effect of the input for some time (for several units of the time constant) until the decaying voltage approaches the resting potential. Hence, a cell with a long time constant (say, 30 msec) will sum successive inputs that arrive every 5 msec (for example) better than a cell with a shorter time constant (say, 5 msec). The voltage change will remain for a longer duration in the cell with the longer τ_m following an input. The other parameter, the input resistance R_m , implies that for a given (long-lasting) input, the cell with the larger R_m value will produce a larger voltage displacement (Fig. 6.2 and Eq. 6.5). Hence, a more powerful input will be required for a similar displacement of V_m in cells with a small R_m value. These important consequences of τ_m and R_m for synaptic integration in nerve cells can be explored using the *Neuron* tutorial.

Finally, it is worth recalling from Chapter 2 that the “membrane model” of Figs. 6.1A and 6.1B serves as a basis for many models that are concerned with the electrical activity of nerve cells. Any additional type of channels (e.g., voltage-gated channels) that may be present in the modeled patch of membrane can be represented by an additional (voltage-dependent) resistive branch and an associated battery, both in parallel to the circuit of Fig. 6.1B. Furthermore, a distributed (non-isopotential) system, such as the dendritic

tree or the axon can, in principle, be constructed from a set of such patches of isopotential membrane compartments that are connected to each other through the cytoplasmic (axial) resistance, as we have done in Chapter 5. This is indeed the common approach used to model the spread of potentials along the detailed structure of dendritic and axonal trees (Segev et al. 1989).

6.3 Response to Activation of Synaptic Channels

In the case of the classical *fast synapse*, the release of neurotransmitter from the presynaptic terminal directly results in the opening of chemically gated ion channels at the postsynaptic membrane. Thus, the input induced by the synapse is primarily expressed as a *local conductance change* at the membrane situated just opposite to the presynaptic release site. Subsequently, specific ions can flow through these channels to produce the synaptic current (I_{syn}) that gives rise to the postsynaptic potential. A concise review of other types of synapses may be found in McCormick (1990).

It is important to note that although both the synaptic input and an electrode which injects a current pulse (as in the case analyzed above) result in the flow of ions across the membrane, the two types of inputs differ in a significant way. Although the electrode does not change the properties of the membrane (provided that the membrane is not injured by the electrode), the synaptic input, due to its inherent characteristics, changes the membrane properties of the postsynaptic cell; it opens new channels there. In the case of an electrode, the current is produced by an external source (the electrode), whereas in the case of a synapse the current source is part of the neuronal system (the ion gradients across the membrane and the transmembrane synaptic channels). As shown below, the difference between these two type of inputs is manifested in the behavior of the corresponding membrane voltage.

6.3.1 The Postsynaptic Current

In Fig. 6.1B, the opening of synaptic channels in an isopotential patch of membrane is modeled by a time-dependent conductance change ($g_{syn}(t)$). This conductance lies in series with a battery (E_{syn} , the synaptic reversal potential or the synaptic battery) that drives the ions involved in the synaptic process. Note that here we assume that the synaptic channels are time-dependent but voltage-independent. This is true for many types of synaptic channels but not for all (e.g., the NMDA receptor, which is modeled in Chapter 19). According to Ohm's current law, the synaptic current through the right-most branch in Fig. 6.1B is,

$$I_{syn}(t) = g_{syn}(t)(V_m - E_{syn}), \quad (6.8)$$

where g_{syn} is the synaptic conductance in siemens. V_m , the voltage across the membrane and E_{syn} , the synaptic battery, are both measured relative to the resting potential (which was set to zero in the present chapter).

When only the synapse is active (without an external current source) V_m is the postsynaptic potential that arises from the activation of the synaptic channels. As can be seen from the circuit in Fig. 6.1B, an increase in g_{syn} causes V_m to move closer to E_{syn} . Therefore, the direction of the change in V_m depends on the sign of the difference ($V_m - E_{syn}$). If E_{syn} is more positive than V_m , then an increase in g_{syn} causes V_m to be more positive — a depolarization. If $V_m > E_{syn}$, the activation of the synapse hyperpolarizes the cell. Note also that when $V_m = E_{syn}$, the activation of the synapse does not produce synaptic current (Eq. 6.8) and thus the voltage across the membrane does not change when such a synapse is activated (a *silent synapse*). However, the activation of such a synapse causes an increase in the input conductance (a decrease in the input resistance) of the postsynaptic cell since it does open new channels there. As a result, the voltage response of such a shunted cell to *other* inputs (either from an electrode or from another synapse) will decrease relative to the case where the silent synapse is not active. This *inhibitory* effect is further discussed below.

6.3.2 The Postsynaptic Potential

As in the case with a current that is injected into a cell through an electrode, the displacement of the voltage at the postsynaptic membrane that results from opening synaptic channels depends on the magnitude and shape of the synaptic current, as well as on the passive electrical properties of the postsynaptic cell. Hence, the voltage (the PSP) that is developed in the circuit of Fig. 6.1B due to the activation of the synaptic input depends both on the characteristics of the synaptic branch (on g_{syn} and E_{syn}) as well as on the passive (g_{rest} and C_m) elements. Again, when no external current is injected into the cell model of Fig. 6.1B, the net current across the membrane is zero. This time it is the sum of the capacitive current I_C , the ionic current that flows through the resting channels I_{rest} and the synaptic current (I_{syn}),

$$I_C + I_{rest} + I_{syn} = 0 \quad (6.9)$$

or:

$$C_m \frac{dV_m}{dt} + g_{rest} V_m + g_{syn}(t)(V_m - E_{syn}) = 0. \quad (6.10)$$

The solution, $V_m(t)$, describes the buildup of voltage while the synaptic channels are opened (i.e., when $g_{syn} > 0$). For the special case where the synaptic conductance change is a rectangular pulse with an amplitude of g_{syn} and a duration of t_{syn} , the solution, obtained as before by separating variables and integrating, can be written explicitly,

$$V_m(t) = \frac{g_{syn}}{g_{syn} + g_{rest}} E_{syn} (1 - e^{-t(g_{syn} + g_{rest})/C_m}), \text{ for } 0 \leq t \leq t_{syn}. \quad (6.11)$$

This equation connects the conductance change induced by the neurotransmitter (g_{syn}) and the transient PSP induced at the postsynaptic membrane. The steady-state solution, obtained when the synaptic channels are opened for an infinitely long duration ($t_{syn} \rightarrow \infty$) is

$$V_m = \frac{g_{syn}}{g_{syn} + g_{rest}} E_{syn} = \frac{1}{1 + g_{rest}/g_{syn}} E_{syn}. \quad (6.12)$$

Equations 6.11 and 6.12 are strictly applicable only to step-conductance changes. Nevertheless, they provide several important general insights into the functional consequences of synaptic mechanisms. One is that, unless $E_{syn} = 0$ (a silent synapse) the PSP is always smaller (in absolute value) than E_{syn} (Eq. 6.12). Only when $g_{syn} \gg g_{rest}$ (the total conductance of the synaptic channels is much larger than that of the resting channels) does V_m approach E_{syn} . Secondly, one can see that V_m is a nonlinear (sublinear) function of g_{syn} . For example, Eq. 6.12 tells us that if $g_{syn} = g_{rest}$, then $V_m = E_{syn}/2$. Assuming that $E_{syn} = 90 \text{ mV}$, the steady-state value of the PSP is 45 mV for this particular example. Multiplying g_{syn} by a factor of two (so that $g_{syn} = 2g_{rest}$) produces a steady depolarization of 60 mV rather than 90 mV , as expected in a linear case. This nonlinearity implies also that successive synaptic inputs (unlike current inputs) will not sum linearly with each other. We examine this nonlinearity in more detail below.

The third point to note from Eq. 6.11 is that, as with a current step (Eq. 6.4), for the case of a rectangular g_{syn} the PSP increases exponentially as long as the synaptic input is present. However, these two cases are markedly different, because in the case of the synaptic input the time constant depends on g_{syn} as well as on g_{rest} . The time constant $C_m/(g_{syn} + g_{rest})$ in Eq. 6.11 is briefer than the resting time constant $\tau_m = C_m/g_{rest}$. Hence, when the synaptic channels are opened, $V_m(t)$ builds up faster than in the resting conditions. The larger g_{syn} is, the faster $V_m(t)$ develops towards its maximal value.

At the end of the synaptic action (for $t > t_{syn}$), when the synaptic channels are closed again ($g_{syn} = 0$), the membrane conductance returns to its passive (resting) value. During this time interval the synaptic potential decays towards the resting value with the passive time constant τ_m , as in the case of the current input (Eq. 6.7). Thus, unlike the case of a constant current input of Fig. 6.2, in the case of a synaptic input the PSP rises faster than it decays.

What happens when several synapses, each with its own conductance change and battery, impinge on the same isopotential patch of postsynaptic membrane? The resulting equation is similar to Eq. 6.10, with the additional synaptic currents added to the sum,

$$C_m \frac{dV_m}{dt} + g_{rest} V_m + g_{syn}^{(1)}(t)(V_m - E_{syn}^{(1)}) + g_{syn}^{(2)}(t)(V_m - E_{syn}^{(2)}) + \dots = 0. \quad (6.13)$$

Now, each synaptic input may have a different reversal potential ($E_{syn}^{(1)}$, $E_{syn}^{(2)}$, ...) and a different corresponding conductance change ($g_{syn}^{(1)}$, $g_{syn}^{(2)}$, ...) which may be activated at different times (Δt_1 , Δt_2 , etc.), respectively. Depending on the sign of the difference, $V_m - E_{syn}$, some synapses may contribute depolarizing currents, whereas others may contribute hyperpolarizing currents. The PSP in this case is the (nonlinear) sum of the effects of all the synaptic inputs. The general solution to Eq. 6.13 for the case where all g_{syn} s are rectangular is an extension of Eq. 6.11,

$$V_m(t) = \frac{g_{syn}^{(1)} E_{syn}^{(1)} + g_{syn}^{(2)} E_{syn}^{(2)} + \dots}{g_{total}} (1 - e^{-g_{total}t/C_m}). \quad (6.14)$$

Here, g_{total} is the sum of all the conductances at that patch of membrane, namely,

$$g_{total} = g_{rest} + g_{syn}^{(1)} + g_{syn}^{(2)} + \dots \quad (6.15)$$

Note that when $E_{syn} = 0$ (i.e., a silent synapse) it contributes to the sum in the denominator but not to the numerator. Thus, such a synapse acts to reduce $V_m(t)$ and is, therefore, called an *inhibitory synapse* (see Sec. 6.4).

6.3.3 Smooth Synaptic Conductance Change: The “Alpha Function”

The synaptic conductance change is better described by a smooth function rather than by a rectangular pulse, as treated above. It is convenient to use an analytical expression to approximate the smooth shape of the experimentally observed synaptic conductance change. A fairly good approximation may be obtained by an analytical function that was first used for this purpose by Rall (1967) and later by Jack et al. (1975) and is referred to as an *alpha function*,

$$g_{syn}(t) = g_{max} \frac{t}{t_p} e^{(1-t/t_p)}. \quad (6.16)$$

This function increases rapidly to a maximum of g_{max} at $t = t_p$. Following its peak, $g_{syn}(t)$ decreases more slowly to zero, as was shown to be the case in the previous chapter. Note that in Eq. 6.16, $g_{syn}(t)$ is determined by two independent parameters g_{max} and t_p . A “slow synapse” (a synapse whose channels kinetics are slow) will be modeled by a relatively large t_p . A powerful synapse (a synapse that opens many channels and produces a significant conductance change) will obtain a large g_{max} value.

The GENESIS simulator uses a slightly more general form, the dual exponential function, to describe $g_{syn}(t)$,

$$g_{syn}(t) = \frac{Ag_{max}}{\tau_1 - \tau_2} (e^{-t/\tau_1} - e^{-t/\tau_2}), \text{ for } \tau_1 > \tau_2, \quad (6.17)$$

where A is a normalization constant chosen so that g_{syn} reaches a maximum value of g_{max} . When $\tau_1 = \tau_2 = t_p$, this is equivalent to the alpha function form of Eq. 6.16. When the synaptic conductance change is modeled by either Eq. 6.16 or 6.17, there is no explicit solution to Eq. 6.10 and the numerical techniques discussed in Chapters 2 and 20 must be employed.

Figure 6.3 shows the response of the cell model in Fig. 6.1B to the activation of a train of four identical excitatory synaptic inputs at 2 msec intervals. The conductance, shown in the lower plot, is of the form given by Eq. 6.16. Here, the rise time of the synaptic conductance is brief, $t_p = 0.2$ msec, whereas the membrane time constant is relatively long, $\tau_m = 5$ msec. The resulting postsynaptic potential (upper plot) illustrates an important aspect of the temporal behavior that was discussed in the previous chapter (Sec. 5.4.3) and which we will explore with the *Neuron* tutorial. As shown in the dashed curve, the much longer membrane time constant causes the PSP due to a single synaptic input to persist much longer than the conductance change. This results in a *temporal summation* of the series of inputs to produce the larger PSP shown in the solid curve.

6.4 A Remark on Synaptic Excitation and Inhibition

As explained below, it is functionally reasonable to define a synapse as being excitatory or inhibitory with respect to the value of the threshold for action potential firing V_{th} . Hence, a synapse that increases the conductance and whose reversal potential is more positive than V_{th} will tend to excite the cell (since it can, provided that g_{syn} is sufficiently large, produce a PSP that is more depolarizing than V_{th}). Thus, the corresponding potential is called an EPSP (*excitatory postsynaptic potential*). A synapse that produces a conductance increase and whose reversal potential is more negative than V_{th} will tend to inhibit the cell from firing. The corresponding potential is called an IPSP (*inhibitory postsynaptic potential*).

Using this operative definition, a synapse with $E_{syn} \leq 0$ (the synaptic battery is more negative than the resting potential) is, clearly, an inhibitory synapse. The activation of such a synapse will both shunt and hyperpolarize the cell. As already discussed, a synapse will only increase the conductance (without changing the voltage) when $E_{syn} = 0$. In both cases the synapse acts to reduce the effect of the EPSPs produced by excitatory inputs that impinge on the cell, thus making the firing of an action potential less likely.

Note that, according to the preceding definition, a synapse may be inhibitory but still produce depolarization. This is true for the case when $0 < E_{syn} < V_{th}$. However, note

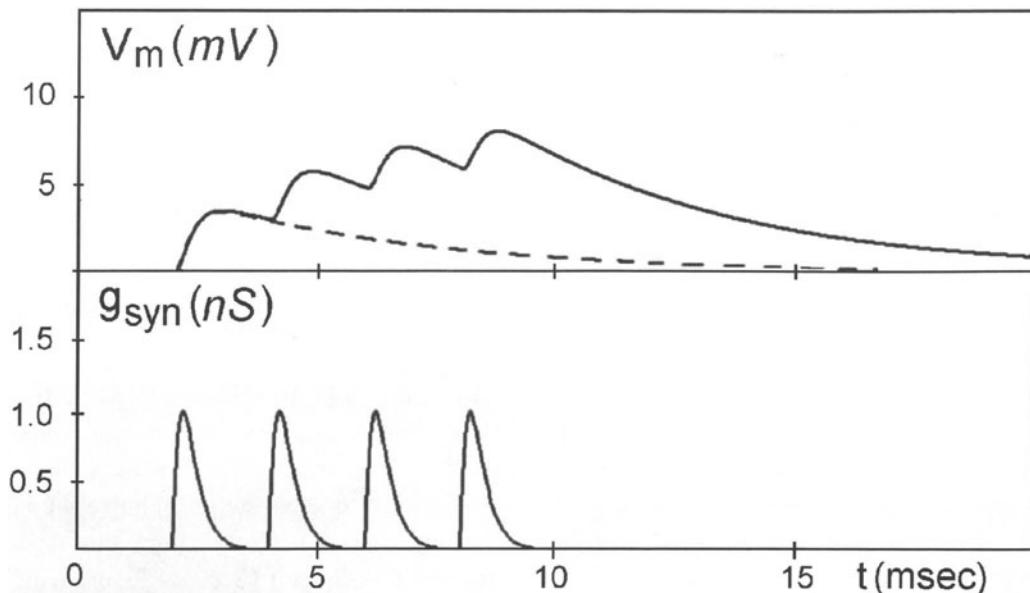


Figure 6.3 The response of the cell model in Fig. 6.1B to the activation of a train of four identical excitatory synaptic inputs at 2 msec intervals. The conductance, shown in the lower plot, is in the form of an alpha function. Each of these inputs has a peak value (g_{max}) of 1 nS and a time-to-peak (t_p) of 0.2 msec; the associated synaptic battery (E_{syn}) is 50 mV above the resting potential. The upper plot shows the potential change (the PSP) for a single synaptic input (dashed curve) and for the series of four inputs (solid curve). As a result of the larger membrane time constant (here, as in Fig. 6.2, $\tau_m = 5$ msec), the PSP persists much longer than the conductance change associated with the synaptic input. This allows a temporal summation of the inputs to produce a larger PSP than would be produced by an individual input.

from Eq. 6.12 that the activation of such a synapse cannot, in principle, reach the threshold for action potential firing. It can be shown that, in general, the facilitative effect of such a subthreshold depolarization is less significant than the inhibitory effect (on excitatory inputs) of the accompanied shunt induced by this synapse. Thus, it is still justified to call such a synapse inhibitory; the corresponding potential is sometimes called a depolarizing IPSP (Segev and Parnas 1983).

6.5 GENESIS Experiments with PSPs

We can study the effects of various synaptic inputs by using the *GENESIS Neuron* tutorial, which was briefly discussed in Chapter 3. This tutorial lets us perform experiments on a simple neuron model consisting of a soma and two dendrite compartments. The soma has Hodgkin–Huxley voltage-activated channels like the ones we used in the *Squid* simulation, and the dendrite compartments have both excitatory and inhibitory synaptically activated

channels that respond to spikes applied at the synapses.

To start the tutorial, change to the *Scripts/neuron* directory and give the command “genesis Neuron.” Once the graphs and “control panel” appear on the screen, click the left mouse button on the box labeled HELP and spend a few minutes exploring the topics on the help menu, starting with Using Help. When you are ready to begin, set the soma injection current to zero in the Soma Inj dialog box. (Don’t forget to hit “Return” after changing the value in a dialog box.) As we will be providing synaptic input to dendrite compartment #1, it would be a good idea to examine the default values of the parameters used for this compartment and the two channels that it contains. Click on Dendrite 1 under the Cell Parameters heading. After reviewing the parameters used to model the compartment, click on Exc. Ch. at the top in order to bring up a secondary menu showing the parameters for the excitatory channel. This should reveal that the ionic equilibrium potential has been set to -10 mV . In this simulation, E_{rest} has been set to -70 mV rather than 0, as we have done so far in this chapter. Thus, $E_{syn} = -10\text{ mV}$ corresponds to a value 60 mV above the resting value of V_m . This potential is typical of that for a depolarizing channel that allows both Na^+ and K^+ ions to pass. Although this channel is called the “Dendrite 1 Excitatory Channel,” the name is due to the default value used for E_{syn} . By setting this to a value less than E_{rest} we could turn it into a hyperpolarizing inhibitory channel.

The default value of g_{max} (given in nS) corresponds to a value of 0.1 nS , or 0.1×10^{-9} siemens. This value is typical for a single synaptic input. The channel model used in GENESIS simulations is of the dual exponential form given in Eq. 6.17. As the two time constants τ_1 and τ_2 are each 3 msec , this corresponds to an alpha function (Eq. 6.16) with $t_p = 3\text{ msec}$. For now, we will use the default values of these parameters, so click on DONE to put this window away.

Clicking on Inh. Ch. in the Dendrite 1 parameter window brings up a similar window for the channel that is nominally the inhibitory channel. The equilibrium potential has been set to -80 mV , which is the value used for potassium ions in our model. The maximum channel conductance and the time constants are the same as those used for the “excitatory” channel. Click on DONE in both windows to put them away. The default parameters for the Dendrite 2 compartment and its channels are the same as those for Dendrite 1.

6.5.1 Temporal Summation of Postsynaptic Potentials

We will start our investigation by applying a train of spike inputs to the excitatory channel of Dendrite 1. This should teach us several principles regarding the temporal summation of EPSPs. A single synaptic input is rarely sufficient to produce a very large effect in most neurons, so this simulation provides a simple way to scale the value of g_{max} without having to call up the sequence of menus needed to change g_{max} . The dialog box labeled Dend #1 Exc. Wt. contains the value of a synaptic weight to be used to scale the synaptic conductance. Thus, entering “10” is equivalent to providing simultaneous input to 10 identical synapses,

or to multiplying g_{max} for this channel by 10. The default value of 0 means that the channel is receiving no input. The toggle to the right of the dialog box can be used to switch the input between Source A and Source B.

For our experiment, set the weight for the Dendrite 1 excitatory channel to 10 and leave the input set to Source A. We will use the default timings for the spike trains that come from Source A. Click on STEP and observe the three plots on the left. The upper plot shows (in red) that Source A is delivering a burst of spikes at 10 msec intervals. The plot below it shows the resulting channel conductance. Note the difference in the rise and fall times for the conductance. Can you explain these? The lower plot shows the rather small increase in the PSP associated with each spike. The plot to the right of this shows similar, but slightly attenuated, changes in the soma membrane potential.

Next, we would like to explore the effect of increasing the rate of input to this channel. This can be done by clicking on the Inputs button in the control panel. The menu that appears contains dialog boxes to set the delay before the onset of the burst of spikes, the width (duration) of the burst, and the interval between spikes. Change the spike interval for both Source A and Source B to 2 msec. In order to easily compare the results with the previous ones, click on the overlay toggle so that it reads Overlay ON, and then click on RESET and STEP.

As with Fig. 6.3, note the temporal summation of the consecutive increases of conductance in the plot of the channel conductance. These result in a buildup of the EPSP, as each increase is added to the previous one. Eventually, the potential becomes large enough to trigger action potentials in the soma. After the train of input spikes ends, the PSP decays to the point where no more action potentials are produced. If you look carefully, you may notice that the action potentials are a little higher in the plot of the soma membrane potential. This is because the PSP in the dendrite is propagated to the soma, where the voltage-activated channels cause the action potentials, and these are propagated back to the dendrite through the axial resistance. In the dendrite compartment, we are seeing a superposition of the PSP produced here and the action potential that is produced in the soma. If we were to look in a more distant dendrite section, the peaks would be much more attenuated.

To examine the effect of adding IPSPs to EPSPs, all we need do is to set the weight for the Dendrite 1 inhibitory channel to 10, so that it will receive input from Source B. We previously set the Source B spike interval to 2 msec, and the default delay is 10 msec after the start of the spike train from Source A. After setting the weight, set the overlay toggle to OFF, click on RESET to clear the graphs, and click on STEP to run the simulation. You should see a result similar to that shown previously in Fig. 3.3. Shortly after the first action potential, we get a large buildup of conductance for the potassium channel. This decreases the net PSP in the soma to a value below that necessary for the production of further action potentials.

In Chapter 5, we studied the passive propagation and attenuation of pulses through a “cable” consisting of many compartments. Here, we can easily make a qualitative examina-

tion of the effect of spatially separating synaptic inputs by performing one more experiment on the model neuron. The simulation gives us the option of putting any number of passive “cable” compartments between the two dendrite compartments by entering a non-zero integer in the **Cable Compts.** dialog box. Set this value to 10 and switch the excitatory input to the **Dendrite 2** compartment using a weight of 100. Use the same input timing as in the previous experiment and compare the results with weights of 0 and 10 for the **Dendrite 1** inhibitory channel. Note how a comparatively small inhibitory input is able to inhibit the much larger excitatory input.

In order to see the channel conductances and the membrane potential in the **Dendrite 2** compartment, click on the **Plot Soma** toggle so that it reads **Plot Dend2**. Now, the lower right graph will plot the membrane potential for **Dendrite 2** instead of the soma. You should see very large EPSPs in **Dendrite 2**, with very attenuated soma action potentials superimposed. From the analysis of Chapter 5, you learned how to calculate this attenuation. Using the values of the parameters given for the dendrite compartments, can you verify that the attenuation is that which would be expected?

6.5.2 Nonlinear Summation of Postsynaptic Potentials

We can learn more about the principles underlying temporal summation of PSPs if we can find a way to eliminate the production of action potentials in the soma. This might be done by blocking the voltage-activated sodium channels in the soma, or by separating a section of dendrite from the soma. In our model, this could be accomplished by setting the maximum conductance of the soma sodium channel to zero, or by greatly increasing (decoupling) the axial resistance between the dendrite compartments and the soma. The latter method has the additional advantage that it reduces the conductance loading of the soma, with its smaller membrane resistance, so that we are left with an isolated section of dendrite and consequently may observe larger PSPs.

We will perform the rest of our experiments on the **Dendrite 2** compartment, isolating it from the rest of the cell. Call up the **Cell Parameters** window for **Dendrite 2** and increase the specific axial resistance R_A to $100\text{ }K\Omega \cdot \text{cm}$. Note the effect on the value displayed for R_{axial} (R_a , in $K\Omega$). In order to analyze our results, we need to know the value of g_{rest} for this compartment. In the notation of this simulation, the input resistance (or membrane resistance) R_m in $K\Omega$ is R_{mem} and the specific membrane resistance R_M in $K\Omega\text{cm}^2$ is RM . You should verify that the values given in the dialog boxes for these two parameters lead to the result that $g_{rest} = 1.26\text{ }nS$. After making this calculation, click on **DONE** to put the menu away. Click on the **Plot Soma** toggle so that it reads **Plot Dend2**, in order to plot the membrane potential for **Dendrite 2** in the lower right graph. Finally, set the overlay toggle to **OFF** and click on **RESET** to clear the graphs.

Let us start by demonstrating one of the important characteristics of synaptic inputs that was discussed above: the inherent nonlinearity in the generation and summation of

PSPs. We would like to apply a single spike input to the excitatory channel of Dendrite 2, using Source A. This may be accomplished by calling up the Inputs menu and setting the Source A spike interval to something large, say, 100 msec. Then set the Dend #2 Exc. Wt. dialog box value to 1, and the weights for the other three synapses to 0. Click on STEP to run the simulation. When it has finished, toggle to Overlay ON, click on RESET, and run the simulation again with a weight of 2. Note that this is equivalent to simultaneously activating 2 synapses, each having $g_{max} = 0.1 \text{ nS}$ or to activating a single synapse with $g_{max} = 0.2 \text{ nS}$. Repeat this experiment with weights of 10, 20, 100 and 200 and then estimate the peak values of the PSPs relative to the resting potential. In order to measure the heights of small PSPs, you can click on the scale button in the upper left corner of the graph. This will bring up a window with dialog boxes that can be used to set y_{min} and y_{max} to appropriate values.

From these experiments, you should conclude that the simultaneous inputs sum linearly for small values of g_{max} , but that the summation becomes increasingly nonlinear as g_{max} increases. We can understand this behavior from Eq. 6.11 and the discussion that follows it. If we approximate the alpha function conductance with a rectangular pulse having amplitude $g_{syn} = g_{max}$ and duration t_p , Eq. 6.11 tells us that the PSP will be roughly proportional to g_{syn} when $g_{syn} \ll g_{rest}$. When $g_{syn} \gg g_{rest}$, the amplitude of the PSP approaches E_{syn} and is independent of g_{syn} . This analysis is not exact, because we have neglected the details of the time dependence of the synaptic conductance. Nevertheless, you should be able to see rough agreement with the measurements that you have performed.

The effect of a “silent inhibition” on the EPSP may be examined using a similar procedure. Use Source B as an input to the Dendrite 2 inhibitory synapse, setting the timing parameters for Source B to be the same as those for Source A. Set E_{syn} for this channel to be equal to E_{rest} , -70 mV , and experiment with different magnitudes of the synaptic weights in order to make the effective value of g_{syn} much smaller or much larger than g_{rest} . What does Eq. 6.14 tell you about the circumstances in which the silent inhibition will have an effect upon the EPSP?

The exercises and projects listed at the end of this chapter suggest several other properties of summed synaptic inputs that you may explore with this simulation. You may also wish to design your own computer experiments in order to gain further insights into the local integration of synaptic inputs.

6.6 Concluding Remarks

We hope that you enjoyed the experiments with postsynaptic potentials and that they have helped to clarify some basic questions regarding synaptic summation and integration. One point to remember is that when membrane channels are the source for the production of voltage changes across the membrane, the reversal potential associated with these channels

is the maximal value that can be achieved using this mechanism. Hence, unless active electrogenic pumps are involved or current is injected from an external source (e.g., by an electrode), the membrane potential of the nerve cell can only vary among the ionic batteries involved (determined by the concentration gradients of the permeable ions across the membrane). In nerve cells the value of the ionic batteries is in the range of -30 mV up to 150 mV or so relative to the resting potential.

Secondly, synaptic inputs are inherently nonlinear inputs since the input itself (the conductance change) perturbs the system (the neuron's membrane). The nonlinearity is more apparent when the conductance change is significant (relative to the resting conductance). The nonlinearity is also marked when the membrane potential V_m is close to the reversal potential of the synaptic process (as is commonly the case with inhibitory synaptic inputs).

Finally, the fact that the neuron with its synaptic inputs is a nonlinear system has several important consequences for the information processing function of the cell. It can be shown that a rich repertoire of operations which could not be implemented in a linear system can be implemented in nonlinear systems such as real nerve cells. You may read more about this important issue in papers by Fatt and Katz (1953), Koch and Poggio (1987), Mel (1993), Rall (1964, 1967), Segev and Parnas (1983), and Segev (1992).

6.7 Exercises and Projects

Unless otherwise noted, perform these experiments on the Dendrite 2 compartment, using a large value of the axial resistance in order to isolate the compartment from the rest of the cell.

1. Find out how changes in the specific membrane resistance R_M in the Dendrite 1 compartment affect the amplitude and area of the EPSP. Check this once with the default value of t_p (3 msec) and then construct a faster synaptic input, say, $t_p = 0.3\text{ msec}$. Which one is more sensitive to changes in R_M ? Explain your results.
2. For $g_{syn} \ll g_{rest}$, is the size of the PSP simply related to the size of the postsynaptic compartment? Is the value of t_p relevant in this case? Perform some experiments to investigate these effects and explain your results. Why is the dependence on t_p much less than in the case of the previous exercise? Suppose that g_{syn} is expressed in terms of a channel conductance density with units of mS/cm^2 , so that g_{syn} scales with the area of the compartment. Would you then expect to see any dependence of the PSP on the size of the compartment?
3. Investigate the conditions under which a depolarizing IPSP can either facilitate or inhibit a simultaneous EPSP. To do this, set E_{syn} for the isolated "Dendrite 2 inhibitory channel" to -65 mV , so that it is 5 mV above the resting potential. Use the

default values for the other channel parameters. Set the input timings for Source A and Source B so that they will each produce a single spike input at the same time. Experiment with both large and small magnitudes of the synaptic weights for the two channels in order to find out which values of g_{syn} will increase or decrease the PSP relative to the case when the inhibitory conductance is zero. Explain your results in terms of the equations given in this chapter.

4. How does the duration of the conductance change (the time-to-peak t_p) affect the amplitude of the PSP? Construct a plot of V_{peak} (measured relative to E_{rest}) vs. t_p and explain why your results are what one would expect.
5. Connectionist artificial neural networks often use a sigmoidal (S-shaped) curve to represent the input-output relation of an artificial “neuron.” The input is an analog value representing the average rate of spikes that would be input to the neuron and the output represents the average firing rate of the neuron. Perform some experiments on this model neuron with the value of R_A and all other parameters restored to their default values. See if a plot of the firing frequency vs. input spike rate has roughly this shape (being zero for small inputs and saturating at a constant value for large inputs). Provide excitatory input to Dendrite 1 with a synaptic weight of 12 and vary the spike interval for Source A.

Chapter 7

Ion Channels in Bursting Neurons

JAMES M. BOWER and DAVID BEEMAN

7.1 Introduction

The simple neuron model that we have built up over the past three chapters is what Llinás (1988) has referred to as the “Platonic Neuron.” In this idealized model, postsynaptic potentials in the dendrites (Chapter 6) propagate passively through the dendritic “cable” (Chapter 5) to the soma. Here, near the axon hillock, the summed and attenuated PSPs may activate voltage-dependent sodium and potassium channels that are very much like those found in the squid giant axon (Chapter 4). Although it was once assumed that this simplified description applied to most neurons, we now know that the situation can be much more complex.

In this chapter we are mainly concerned with the additional varieties of ionic conductances that are responsible for the wide variations in firing patterns seen in different neurons. Toward the end of the chapter, we examine a model that illustrates another difference between the Platonic neuron and many mammalian neurons. In many neurons, the dendrites are not merely passive conduits for the propagation of PSPs, but are endowed with a variety of conductances that can actively shape and amplify neuronal signals (Llinás 1988, Adams 1992).

In Chapter 4 we considered the ionic conductances underlying the generation of the action potential in the squid giant axon. This description of transmembrane sodium and potassium conductances constituted the first comprehensive and quantitative understanding of the ionic events associated with neuronal activity. As has often been the case in the history

of biology, this advance in understanding was dependent on the development and application of a new technology, the voltage clamp. However, equally important was the choice of the squid giant axon as the subject of the experiments.

This experimental preparation was chosen by Hodgkin and Huxley principally because the giant size of the squid's axon made the insertion of multiple electrodes possible. However, there is another reason that the choice of the squid axon was fortuitous. It turns out that the conductances found in this axon are fewer and more simplified than those found in any other region of a typical neuron. The reason is that axons, in general, are highly customized neural structures. In the squid, the giant axon is solely responsible for assuring the rapid and regular conduction of the neural impulse to the muscles of the squid's mantle. The repeated simultaneous contraction of these muscles, in turn, provides the force behind the animal's water-jet propulsion system.

Because the sole function of the squid giant axon is to conduct neural impulses in a rapid and highly regular fashion, the ionic conductances found in this axon's membrane are few and functionally streamlined. However, a very different situation holds for those neuronal regions, like the dendrite and cell body or soma, responsible for receiving and processing information and generating the patterns of activity distributed by the axon. Membranes in these regions often contain a much larger set of ionic conductances. Furthermore, the individual properties of these conductances are often customized to support the function a particular cell plays in the network in which it is found. As we show, these conductances can be quite rich and their interactions quite complex.

In general, the physiological behavior of a neuron, i.e., how it responds to input and generates output, is determined by the types of conductances found in its membranes and the interaction between them. In this chapter we specifically consider a set of ionic conductances that interact to produce periodic bursts of action potentials. The data presented are taken from experiments on the somata of periodically bursting molluscan neurons including those found in the sea slugs *Aplysia* and *Tritonia* as well as neurons in other related marine molluscs. As in the case of the squid giant axon, these cells have been extensively studied because their size and geometry made voltage clamp experiments considerably easier to carry out. Furthermore, individual cells are also readily identifiable from individual to individual allowing the experimenter to use multiple preparations to characterize each cell's properties (Frazier, Kandel, Kupfermann, Waziri and Coggesshall 1967). Although it was once believed that these varied molluscan ionic conductances were relevant only to the study of invertebrates, we now know that they are also found in mammalian neurons (Llinás 1988). Thus, single molluscan neurons can teach us much that is relevant to the understanding of the human brain (Kandel 1976).

As we shall see, different molluscan neurons make use of differences in the densities of these various types of channels to produce a wide variation in the shape and firing patterns of their action potentials. Variations in the voltage thresholds and time constants for activation and inactivation of these conductances are also used to produce differences in behavior.

For more complicated vertebrate neurons, variations in neuronal behavior are accomplished by differences in the distributions of these conductances over the soma and dendrites, not generally found in invertebrates.

We begin with a survey of the ionic conductances found in molluscan pacemaker neurons and a discussion of their effect upon the observed patterns of action potentials. Then, we make use of a GENESIS simulation of a particular bursting molluscan neuron in order to further understand the roles of these conductances. In the final section of this chapter, we use a GENESIS recreation of the Traub, Wong, Miles and Michelson (1991) hippocampal CA3 region pyramidal cell model, which has a similar set of conductances. With the tutorial simulation, we demonstrate how the burst firing of this mammalian neuron depends on the way that these conductances are distributed over the soma and dendrites.

7.2 General Properties of Molluscan Neurons

Individual molluscan neurons, like neurons everywhere, often have very different physiological properties. In any particular case, these properties are related to the information-processing role of the neuron within the animal. Neurons that are found in the abdominal ganglion of *Aplysia californica* are illustrative of the range of firing patterns that are found. Some cells have a stable resting potential and are silent in the absence of synaptic input. Others are never at rest and fire spontaneously. Those that discharge action potentials regularly and continuously at low frequencies, with a pattern like that shown for the *Aplysia* R3 neuron in Fig. 7.1A, are called *beaters*. *Regular bursters*, such as the R15 cell, generate a very regular pattern of bursts of action potentials, as shown in Fig. 7.1B. Still others, such as the L10 cell, generate irregular patterns of bursts, as shown in Fig. 7.1C.

In some cases, the periodic firing can be shown to be entirely endogenous to the neuron itself, and to persist even when the soma is isolated from the rest of the cell. These neurons have regular firing patterns in the absence of input and are often referred to as *pacemakers*. They are almost always associated with the control of behavior that is highly regular and subject to only moderate amounts of external regulation (e.g., respiration). In other neurons, varying amounts of not necessarily periodic input will generate a periodic neuronal output. These types of neurons are usually referred to as *conditional bursters* and tend to be associated with behaviors that are variably periodic but under considerable external control (e.g., swimming). Depending on modulation by synaptic input, they may be silent, beating, or bursting. The distinction between endogenous and conditional behavior is not always clear-cut. For example, the R15 neuron is usually called an *endogenous burster* because it fires spontaneously in the isolated soma, in the isolated ganglion, and in the isolated nervous system. However, recent experiments indicate that it is largely silent in the intact animal and might better be considered as a conditional burster (Alevizos, Weiss and Koester 1991).

The action potentials shown in Fig. 7.1 are produced primarily by channels that are

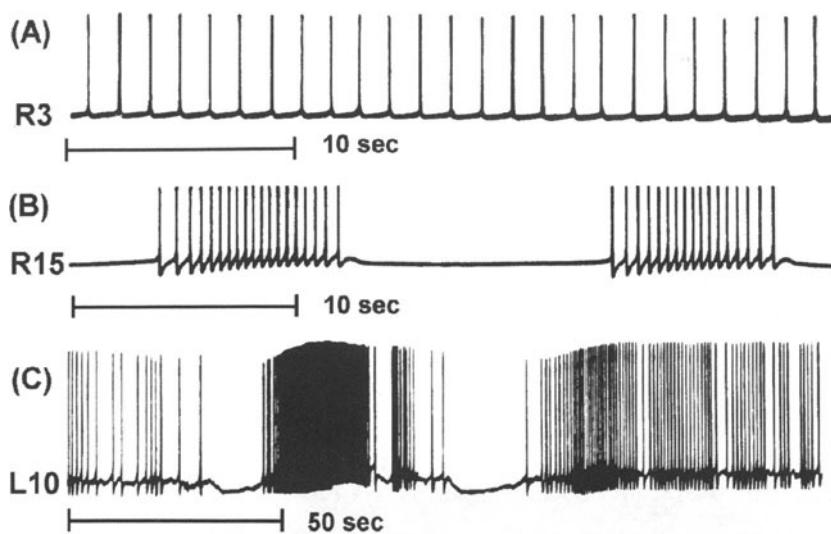


Figure 7.1 Some typical firing patterns of neurons found in the abdominal ganglion of *Aplysia*. (A) The R3 beater. (B) The R15 regular burster. (C) The irregularly bursting L10. (Adapted from *Cellular Basis of Behavior* (Kandel 1976). Copyright 1976 by W. H. Freeman and Sons. Used with permission.)

qualitatively similar to the sodium and potassium channels discussed in Chapter 4. As we shall see, the details of their shape and firing patterns are governed by other ionic conductances that are not present in the squid giant axon. Whether a particular neuron is a silent cell, beater, regular burster, conditional burster, or non-periodic is largely dependent on the particular combination of ionic conductances found in its cell membrane and on the interaction between them. Accordingly, understanding a cell's behavior is dependent on determining the types of conductances it contains, and the density of these conductances.

As mentioned in the previous section, the conductances found in parts of the neuron responsible for determining the overall input-output characteristics of the cell are considerably more complex than those found in its axon. This increased complexity is manifest in numerous ways. For example, these regions of a neuron often include conductances for ions other than sodium and potassium, including chloride and calcium. Second, the activation and inactivation properties of these conductances can be considerably more complex than those seen in the axon. In some cases, the activation and inactivation characteristics can not be adequately fit with the relatively simple exponential and sigmoidal functions used in the Hodgkin-Huxley model. In our model, this has necessitated the use of tabulated forms for some of the activation curves, rather than fits to an analytic function. Third, particular conductances can be voltage-dependent, dependent on the binding of particular molecules to the membrane, temperature-dependent, or a combination of all three. Fourth, the pharmacology of somatic ionic channels is often more complex, considerably compli-

cating experimental procedures. Nevertheless, as we show, a considerable amount is known about the ionic properties of molluscan neurons and the relationship of these properties to the function of individual cells. Finally, and most importantly, the temporal properties of specific conductances also vary considerably, with some having time constants in the millisecond range and others operating over minutes or even hours. These combinations of fast and slow time courses are responsible for the complex dynamics observed in bursting neurons.

7.3 Ionic Conductances — The Dance of the Ions

In this section we give a general overview of the principal ionic conductances that have been found in different molluscan neurons. As we demonstrate, different neurons contain different combinations of these conductances depending on the function they perform. Here, we describe a sodium conductance, a calcium conductance, a combined sodium-calcium conductance and three potassium conductances, one of which is dependent on calcium. Later, in Sec. 7.4.3, we give some details of the specific examples of these conductances that have been incorporated into our model molluscan burster. These conductances, listed on page 115 in Table 7.1, have been taken from various experimental measurements, and are representative of the conductances discussed in this section. Further details of these conductances may be found in the review article by Adams, Smith and Thompson (1980) and the many references given there. Although this is by no means a complete list of the conductances found in molluscan cells, considering the interactions between these primary conductances should provide some insight into how the overall behavior of a cell is influenced by the conductances its membrane contains.

Figure 7.2 presents steady-state activation and inactivation curves for each of the conductances used in the model. Chapter 4 has discussed the procedure by which these curves are obtained from voltage clamp experiments. In Part II of this book, Chapter 19 describes how one uses these experimental data to implement these channel models in GENESIS. In this chapter we consider the significance of their shape for cell function. In this regard it is important to note that the steepest changes in voltage dependence shown by the curves for any one conductance often tend to occur over a relatively narrow range of voltage. Furthermore, the values of steepest dependency vary for different conductances. As we will see, by comparing the region of largest change in activation and inactivation voltages with the observed action potentials, it is often possible to predict which aspect of neuronal signaling a particular conductance most influences. We will also see that the time constants for activation and inactivation of the various conductances have a great influence on the shape of the action potentials and the firing patterns. These have been listed in Table 7.1 near the nominal resting potential of the cell (-40 mV) and at a point during an action potential (0 mV). In this chapter, we use the variable X to represent the state variable for activation,

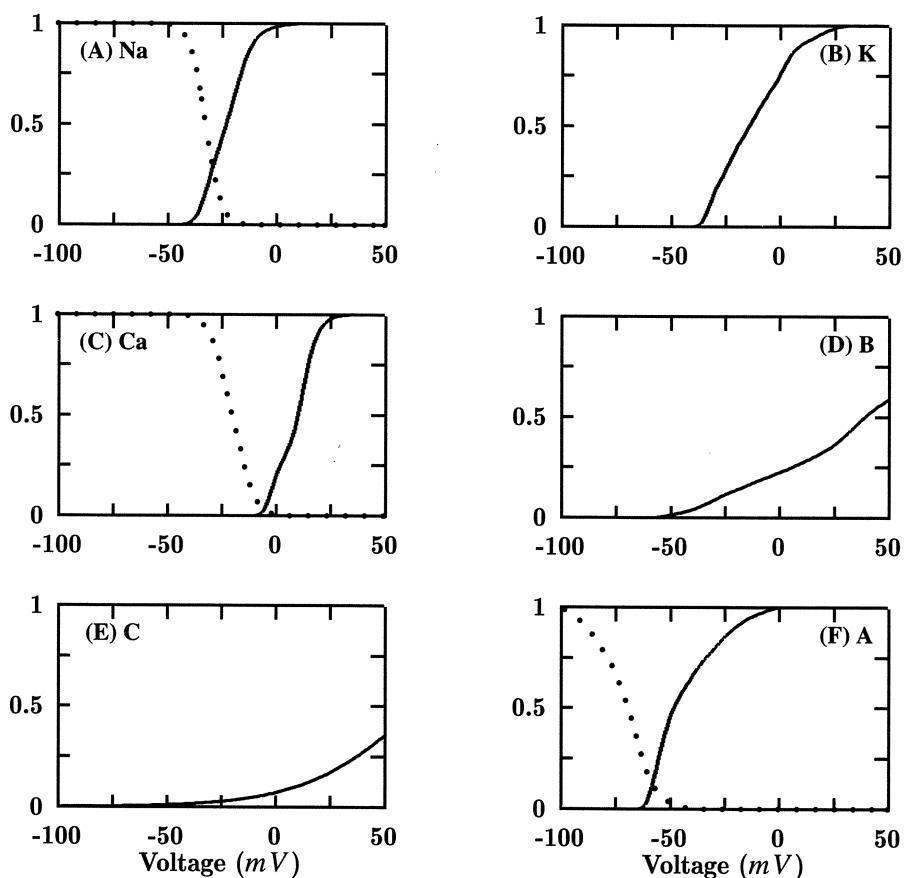


Figure 7.2 Steady-state activation variables (solid lines) and inactivation variables (dotted lines) for the conductances used in the bursting molluscan neuron model. (A) Fast sodium current. (B) Delayed potassium current. (C) High threshold calcium current. (D) Slow inward “B-current.” (E) Calcium-dependent potassium “C-current,” with $[Ca^{2+}]$ held at 187 nM. (F) Transient potassium “A-current.”

and Y to represent the state variable for inactivation. In the case of the squid axon sodium channel, these would correspond to the variables m and h , used in Chapter 4.

7.3.1 Action Potential Related Conductances

The first three conductances to be considered in this section are all involved directly in the voltage changes associated with the action potential. As we will see, action potentials generated in molluscan somata are dependent on sodium and potassium associated ionic

conductances that are similar, although not identical, to those described in the squid axon by Hodgkin and Huxley. In addition, however, many molluscan neurons also include an action potential related calcium conductance. Although not seen in the main body of the squid axon, this conductance is similar in many ways to one found in its terminal region (Llinás, Steinberg and Walton 1976).

Fast Sodium Conductance

The somata of action potential generating molluscan neurons include a rapidly activating sodium conductance. As can be seen in Fig. 7.2A, the region of largest voltage-dependent change in this channel's activity is at and above the threshold for generation of the action potential. Thus, in the molluscan soma, as in the squid axon, this conductance is associated with the generation of the action potential. In both soma and axon this sodium conductance is also generally described by $X^p Y^q$ kinetics and is usually blocked by TTX, suggesting that the conductances are homologous.

Although the axonal and somatic sodium conductances are similar, there are important physiological and pharmacological differences between the two indicating that the channels on which they are based are not identical. One of the differences between these conductances is that a prolonged period of hyperpolarization does not inactivate the somatic conductance as it does in the axon. The result is that many molluscan somata do not generate the “anode break” phenomena briefly discussed in Chapter 4. In Exercise 5, you have an opportunity to discover what features of the activation and inactivation kinetics account for this difference. In addition, the exponent associated with the activation term X can vary in molluscan somata. In fact, differences in activation kinetics are often seen between different molluscan neurons of the same species, resulting in faster action potential rise times in one neuron compared to another. Similarly, the actual voltage dependencies of the activation and inactivation curves can vary from cell to cell. This variation tends to be correlated with differences in the normal resting membrane potential of a particular neuron, with shifts in the positive direction in cells with higher resting potentials and vice versa. In this way, this classic sodium conductance is often customized to the characteristics of a particular neuron.

Delayed Potassium Conductance

In addition to the sodium conductance, the potassium conductance of the squid axon also has a homologue in the molluscan soma. Evaluating the activation curve for this conductance (Fig. 7.2B) suggests that this somatic conductance is also associated with the action potential's falling voltage phase. Thus, as in the squid axon, this potassium conductance is also responsible for repolarizing the membrane after sodium influx. Table 7.1 shows that, as with the squid axon, the time constant for activation of the potassium current is much larger than that for the sodium current. For this reason, it is often referred to as the *delayed* or *late*

potassium current. As the channel conductance is much greater with depolarizing voltages than with hyperpolarizing voltages, it is also called the *delayed rectifier* current.

However, as with the sodium conductance, this potassium conductance also differs from the axonal version in several ways. We may see some of these differences by comparing the steady-state activation curve for this current in a typical mollusc (Fig. 7.2B), with that in the squid axon (Fig. 4.9). Whereas the potassium activation increases sigmoidally, beginning at voltages below the resting potential in the squid axon, the molluscan activation curve starts much more abruptly, beginning at a point above the action potential threshold. Table 7.1 shows that the exponent for the activation parameter is also different, with the conductance being proportional to X^2 for the molluscan potassium current, instead of X^4 .

Another important variation concerns the presence of a voltage-dependent inactivation, which has been observed in the dorid molluscs *Anisidoris* and *Archidoris* (Connor and Stevens 1971a, Aldrich, Getting and Thompson 1979a). This inactivation is more complex than that modeled by Hodgkin and Huxley for the sodium conductance, as the recovery from inactivation is at least an order of magnitude slower than the onset. This effect is often referred to as frequency-dependent inactivation because, as a neuron fires repetitively, the baseline inactivation of the potassium conductance increases. The buildup of inactivation occurs with a relatively long time constant and is slowly reversed when the neuron stops firing. Some decline in potassium current has also been observed in *Aplysia* during long periods of depolarization. However, it has been suggested that in *Aplysia*, the effect is largely due to the accumulation of potassium in the extracellular solution at the membrane surface, reducing the potassium equilibrium potential (Adams and Gage 1979a).

High Threshold Calcium Conductance

One of the ways in which the region of the squid axon studied by Hodgkin and Huxley is particularly unusual as neural tissue is in the complete absence of a calcium conductance. However, the contribution of the squid to membrane biophysics has not been limited to use of its giant axon. In addition, the giant synapse that activates the giant axon has also been used to quantify the properties of a particularly important and ubiquitous calcium conductance (Llinás et al. 1976). This conductance is activated by the occurrence of an action potential and results in the influx of calcium that directly triggers transmitter release.

Because calcium has not been considered before in these chapters, and because it occupies a special place in cell function, we briefly consider the special role played by this ion and the experimental consequences. Calcium conductances, in general, are an important property of the membranes of not only most neurons but also of a wide range of other living cells. The influx or intracellular release of calcium is used quite universally as a trigger for important cellular functions. Cell cleavage during development is triggered by calcium. Calcium is also involved in many fundamental cellular properties through its role in regulating numerous metabolic enzymes.

Beyond its general effects on cell physiology, calcium also plays a host of important specific roles in neurons. Perhaps one of the best known involves the calcium triggered release of neurotransmitter from the presynaptic axon terminal (Katz and Miledi 1969). This calcium conductance underlies the neural machinery involved in transferring information from one cell to another. Of more relevance to this discussion, calcium conductances are also found in the somata and dendrites of many different types of neurons. In some cases these conductances are quite small, whereas in others, membrane voltage changes associated with calcium entry can rival sodium in its responsibility for generating action potentials.

Although calcium is an exceedingly important contributor to neuronal function, it is also typically very difficult to study. In the axon terminal, for example, the minute size of the calcium conductance makes its isolation very difficult even with voltage clamp techniques. In other cases calcium conductances can have long and complex time courses. In addition, the study of pure calcium entry is usually confounded by potassium conductances that are quite difficult to completely block and can largely mask calcium ion flow. Finally, the very close relationship between intracellular calcium and many fundamental metabolic pathways makes it difficult to perform calcium ion substitutions, which are an important experimental approach to isolating the ions associated with different membrane conductances. Ion channel biophysicists must often be at their most clever and careful when studying calcium.

Here, we examine one of the several calcium conductances that are known to exist in molluscan somata. Although its kinetics are much slower, the *high threshold* calcium conductance is qualitatively similar to the sodium conductance in that it inactivates with a time constant that is much greater than that for its activation. Looking at the voltage-dependent activation and inactivation curves for the *Aplysia* R15 calcium conductance (Fig. 7.1C), it can be seen that the steepest region of its voltage dependence is associated with peak voltage values in the action potential. As a result, these calcium channels are maximally activated during and immediately following the action potential's peak voltage, and are not activated in the absence of action potentials. Because of the delay in activation, the calcium channels are maximally activated during the action potential's negative going phase. Because the equilibrium potential for calcium is very positive, this means that there is a tremendous driving force for calcium entry into the cell. Voltage clamp data in the giant synapse also suggest that calcium entry occurs during the falling phase of the action potential. It turns out that this high threshold action potential associated calcium conductance has a homologue in many neurons, including mammalian as well as molluscan neurons (Llinás 1988).

Although the threshold for activation of this conductance is generally above that for the activation of the sodium conductance, there is a large variation in the threshold among different classes of molluscan neurons. For example, it begins to activate at potentials as low as -40 mV in the dorid molluscs (Aldrich, Getting and Thompson 1979b). The physiological effect of this lower threshold appears as a “shoulder” on the downward slope of the action potentials in these neurons. This effect is examined in Exercise 6. Neurons in dorid

molluscs that have this shoulder in the action potential also show a progressive frequency-dependent broadening of the action potentials. This arises as a result of an interaction between the calcium conductance and the frequency-dependent inactivation of the potassium conductance, also observed in these neurons. The longer period of depolarization provided by the shoulder allows the potassium conductance to inactivate, further increasing the width of the shoulder with each successive action potential.

7.3.2 Control of Bursting Properties

The trace of a typical R15 bursting pattern shown in Fig. 7.1B reveals some interesting features which we would like to explain from the properties of ionic conductances that are found in this cell. Typically, bursts are characterized by groups of 2–20 spikes, separated by long (5–30 second) postburst hyperpolarizations. During a burst, the spike frequency first increases and then decreases, and for some cells a plot of the interspike interval as a function of time is approximately parabolic (Strumwasser 1965). There is a slow depolarizing bump after the last spike in a burst, and the bursts appear to be riding on a slow wave of depolarization. In fact, when the sodium conductance is blocked with the neurotoxin TTX and the action potentials cease, slow oscillations are seen that have a period approximately equal to that of the bursts (Strumwasser 1968). Wilson and Wachtel (1974) conducted voltage clamp experiments on a variety of bursting neurons in *Aplysia*. When the steady-state total channel current is plotted as a function of clamping voltage (using the inward negative convention), the resulting curve shows a *negative slope region* (NSR). They proposed that this negative resistance characteristic is essential for the existence of the slow oscillation and burst firing.

It turns out that regulation of the bursting properties of these cells involves additional conductances that control how, when, and for how long neurons are active. In general, control of input-output relationships of individual neurons is largely associated with potassium conductances. The result is that the diversity of neuronal activity patterns is directly paralleled by the diversity and complexity of intrinsic potassium conductances. In the final part of this section we consider two of these conductances.

There are several different mechanisms and conductances that can lead to this sort of oscillation in the firing. In general, we require an inward current with a long time constant in order to produce the sustained depolarization that initiates and maintains the burst. We need another mechanism to terminate the burst and provide a long, but finite, period of hyperpolarization between the bursts. As these two processes involve time scales that are much slower than those for the action potential related conductances described earlier, they may be studied in isolation from these faster conductances by observing the slow *tail currents* that are seen in voltage clamp experiments after the initial rapid changes in current (Smith and Thompson 1987). On the other hand, the slow currents are generally much smaller in magnitude, and are also difficult to separate from each other. For this reason, there is a great

deal of uncertainty as to the ionic basis of these currents and in the measured activation curves.

The slow inward current discussed below is often called the “B-current,” or “burst current.” The basis of the slow hyperpolarizing current has been particularly controversial. Several researchers have concluded that it is a potassium current which becomes activated by the buildup of intracellular Ca^{2+} ions during the burst (Gorman, Hermann and Thomas 1982, Smith and Thompson 1987). This current is often called the “C-current.” At some point, the C-current polarizes the cell below the threshold for the production of action potentials and the burst terminates. During the hyperpolarized interval after the burst, no calcium ions enter the cell and the Ca^{2+} concentration begins to slowly decay, reducing the C-current. The cell then slowly depolarizes, activating the B-current and leading to a new burst of action potentials.

However, other researchers have found that the slow hyperpolarizing current is insensitive to the external K^+ concentration and does not reverse at the expected potassium equilibrium potential (Adams and Levitan 1985, Kramer and Zucker 1985b). They concluded that it cannot be a potassium current and attributed it to the inactivation of an inward calcium current. It turns out that both mechanisms are present, but that the C-current dominates at low temperatures (10–15 °C). At higher temperatures, the C-current is still present, but decays much more rapidly than the potassium-insensitive current (Thompson, Smith and Johnson 1986). It is often the case that, when nature can find two ways to accomplish the same purpose, both methods are used. The *Aplysia* must function in a variety of environments ranging from the cold ocean floor to warm tidepools, so the two different currents provide a way to maintain burst production over a wide temperature range. As the inactivation of the slow inward current has not been as thoroughly investigated or quantified as the calcium-activated potassium current, we concentrate on the currents that dominate at low temperatures in the rest of this section and in our model. A description of the hypothesized currents and bursting mechanisms at room temperature may be found in the review article by Benson and Adams (1989).

Finally, we should briefly mention another mechanism for burst production that is found in some mammalian neurons. The transient low threshold calcium current, or T-current, is found in neurons in many parts of the mammalian nervous system, including the thalamus, inferior olive, superior colliculus, and medial septum, as well as other regions of the brainstem and forebrain. This current is qualitatively similar to the high threshold calcium current discussed in Sec. 7.3.1, but has a much lower threshold for activation and inactivation. Consequently, a small depolarization from a hyperpolarized level can activate the T-current, producing a broad hump that is above the threshold for generating sodium spikes. After activation, it inactivates and cannot be reactivated until the cell is hyperpolarized again. The result is that a burst of action potentials is generated which rides upon the hump (McCormick 1990, McCormick, Huguenard and Strowbridge 1992). For larger depolarizations from the resting potential, the T-current is completely inactivated. In this

case, the cell will either be silent, or if the depolarization is large enough, tonically fire trains of single spikes. In thalamic relay neurons, these different modes of firing are used to gate the sleep-wakefulness cycle. During drowsiness and slow wave sleep, the cells are hyperpolarized and fire in bursts. This results in synchronized cortical oscillations which may be observed in electroencephalograms. Increased excitatory synaptic input from the brainstem and hypothalamus occurs during periods of wakefulness, switching the neurons into the single spike firing mode. In this case, the cells serve as relay elements carrying information to and from the cortex (Llinás 1988, McCormick et al. 1992). Chapter 17 provides the information that will allow you to create your own simulation of a thalamic relay cell.

In the final section of this chapter, we examine yet another mechanism for generation of bursts in mammalian neurons. Currents that are very similar to those found in molluscan bursters produce burst firing in hippocampal CA3 region pyramidal cells. However, in this case, the mechanism for burst production depends upon the structure of the cell and the fact that there are different densities of the various conductances in the soma and dendrites.

Bursting Conductance

Molluscan neurons that burst endogenously often have a specific burst-related conductance. When one of the action potentials from such a neuron is observed in isolation, it can be seen that the action potential is followed by a large depolarization. This can be clearly seen after the final spike in one of the bursts shown in Fig. 7.1B. This depolarizing after-potential is associated with the B-current described here. This slow inward current activates over a period of about one second.

The B-current is difficult to measure because of its small amplitude and because its time course is similar to the decay of the outward C-current. It has been variously attributed to a current of sodium ions (Barker and Smith 1978), calcium ions (Gorman et al. 1982), or a combination of the two (Smith and Thompson 1987). In bursting *Tritonia* neurons, the amplitude of this current is reduced by about 50% when the extracellular solution is replaced by one having one tenth the normal concentration of Na^+ ions. A similar result occurs when the Ca^{2+} concentration is reduced instead. When both ions are absent, the current vanishes. This suggests that in these neurons, Na^+ and Ca^{2+} ions make approximately equal contributions to the B-current (Smith and Thompson 1987). It is also possible that this current arises from two separate ion-specific conductances that have similar time and voltage dependencies. Kramer and Zucker (1985a) found that in *Aplysia* bursting pacemakers L2 through L6, the slow inward current is activated by intracellular calcium. However, these experiments in *Tritonia* showed a voltage dependence that is inconsistent with activation by calcium. Thus, it is possible that the mechanism of activation of the current and the ionic selectivity may be different in the various molluscan bursting pacemaker neurons which have been studied.

Figure 7.2D shows the voltage dependence of the channel activation, taken from measurements of the B-current in bursting *Tritonia* neurons by Smith and Thompson (1987). It can be seen that the B-current begins to be activated in this cell at about -50 mV , well below the point at which the fast sodium conductance is activated. The activation continues to increase throughout the rising phase of the action potential. Taking into account the delay in the conductance onset, it is not surprising that the physiological effect of this conductance is to produce a post-action potential depolarization of the cell soma. Although the total conductance of this channel is about 1% of that associated with the sodium channel, because it is active throughout the voltage range of the action potential and is also active when the cell membrane is at relatively high resistances, this conductance can produce large transmembrane voltage effects.

As mentioned before, the principal physiological effect of this conductance is to depolarize the neuron after the action potential. In fact, this post action potential depolarization alone is often sufficient to produce a second action potential in the cell. Accordingly, once the cell is depolarized, this conductance makes the pacemaker capable of generating repetitive firing with little or no additional input. Furthermore, Table 7.1 shows that the time constant for activation of this conductance is much smaller during an action potential (240 msec at 0 mV) than it is at more polarized levels (1.2 sec at -40 mV). Thus, the B-current increases comparatively rapidly during an action potential, but decays slowly between action potentials, becoming progressively larger with each action potential. Combined with the action potential related conductances discussed in the previous section, this should result in patterns of continuous firing. However, as is clear from Fig. 7.1B, these intrinsically driven bursting patterns eventually terminate. Next, we consider one of the mechanisms responsible for the termination of the burst.

Calcium-Activated Potassium Conductance

In addition to the slow inward current described above, voltage clamp experiments on bursting neurons in *Aplysia* and *Tritonia* show an outward tail current that slowly decays over many seconds after repolarization from a depolarizing voltage step (Thompson et al. 1986, Smith and Thompson 1987). At cold temperatures ($10\text{--}15\text{ }^{\circ}\text{C}$), the amplitude of this current depends strongly upon the extracellular concentration of K^+ ions. When the K^+ concentration of the extracellular solution is increased in order to shift the potassium reversal potential to the clamping voltage, the current may be eliminated entirely, indicating that it is a potassium current. By performing voltage clamp experiments at a variety of extracellular K^+ concentrations, it is possible to separate this current from the B-current, which is insensitive to the external K^+ concentration (Smith and Thompson 1987).

The apparent voltage dependence of the activation of this current is unusual in that it is bell-shaped, rising to a peak at about $+40\text{ mV}$ and then decreasing with further depolarization. The explanation for the observed activation properties has to do with the fact that this

conductance is calcium-dependent. Although the channel associated with this conductance is selectively permeable to potassium, channel opening appears to require the presence of intracellular free calcium. The associated current is often called the C-current. As the cell is depolarized, calcium ions enter via the B-current, activating the C-current. However, as the clamping voltage increases towards the calcium reversal potential, the inward flow of Ca^{2+} decreases, causing a drop in the activation. The calcium dependence of this conductance may be verified by experiments that prevent the intracellular concentration of Ca^{2+} from increasing during depolarization. For example, the substitution of Co^{2+} , a Ca^{2+} current blocker, for Ca^{2+} in the external solution suppresses the C-current (Smith and Thompson 1987).

Gorman and Thomas (1980) performed voltage clamp experiments on *Aplysia* R15 neurons in which Ca^{2+} ions were directly injected into the soma through micro-electrodes. The resulting concentration of free intracellular Ca^{2+} was determined from changes in the absorbence of light by the Ca^{2+} sensitive dye arsenazo III. These experiments revealed that the C-current varies linearly with internal Ca^{2+} over a wide range of concentrations. The intrinsic voltage dependence of the activation of this conductance was observed by using Ca^{2+} injection to hold the internal Ca^{2+} concentration constant. This resulted in the exponential voltage dependence seen in Fig. 7.2E. A comparison of this activation curve with that of the late potassium conductance shown in Fig. 7.2B reveals some significant differences between the two conductances. Rather than having a rapid rise in activation after a fairly sharp threshold for activation, the activation curve for the C-current increases gradually with voltage, with no clear threshold for activation. This suggests that the two conductances may arise from distinct populations of ion channels.

The very slow variation in the C-current after depolarizations or action potentials contributes to spike frequency adaption as well as the long period of postburst hyperpolarization. Both of these have a powerful effect upon the integrative function of the neuron. Because this current, which is also prevalent throughout the mammalian nervous system, depends strongly on the internal Ca^{2+} concentration, it has the potential to be modified by factors that affect cellular metabolism, such as hormones or synaptic modulators (Adams et al. 1980).

Transient Potassium Conductance

Finally, we consider one additional potassium conductance that also influences the action potential generating properties of molluscan as well as many other species of neurons. The associated current is known as the A-current and is found throughout the nervous system. In addition to being kinetically separable from the delayed rectifier, the A-current is also pharmacologically distinct. It is interesting to note that this conductance is actually as ubiquitous as the delayed rectifier in neurons. In fact, the principal reason that it is not as well known as the other potassium conductance is that it does not occur in the squid axon. It has been suggested that this conductance is actually the phylogenetically oldest conductance

and that it is the precursor for the sodium channel (Hille 1984). This conductance has also been isolated to a single gene locus in the fruitfly, *Drosophila*. The *shaker* mutation of this gene prevents the A-current from inactivating, resulting in the prolongation of action potentials and causing the fly to shake violently when it is anesthetized.

Table 7.1 reveals that unlike the delayed potassium current, which inactivates very slowly or not at all, the A-current inactivates moderately rapidly (235 msec) after a fast (12 msec) activation. Thus, its response to a depolarizing pulse is a short-lived outward flow of current. For this reason, it is often called the “transient outward current.”

Figure 7.2E displays the voltage dependencies of activation and inactivation of this conductance and illustrates another significant difference from the conductances that we have considered so far. Unlike the previously described conductances, its steepest voltage dependencies are found in a voltage range near or below resting membrane values. This fits with the role of this conductance in regulating the threshold behavior of the neuron. At the resting potential for the neuron the activation curve for this conductance is quite steep and the time constant for activation is very short. Because the equilibrium potential for potassium is near the resting potential of most neurons, the effect of this conductance is to resist any rapid depolarization of the neuron. As a consequence, transient inputs do not result in the generation of action potentials. However, a sustained input above the resting potential causes the A-current to inactivate, allowing the cell to be driven to the threshold for firing. Note that it is the delayed potassium current, however, which is responsible for the repolarization that follows an action potential. This is because it is strongly activated (with a delay) during the peak of the action potential. At these high potentials, the A-current is completely inactivated and makes no contribution until the cell has become hyperpolarized.

The low thresholds for activation and inactivation of the A-current are responsible for another important function of this conductance. During the hyperpolarization following a spike, the potential is such that the steady-state activation of the delayed potassium current would be zero, so it is decaying with a time constant of about 60 msec. When the A-current is present, it will be activated at these hyperpolarized potentials. Thus, it will produce an outward current that maintains the hyperpolarization, slowing the onset of the next action potential. This allows the cell to produce the slow firing that is typical of pacemaker cells. With increasing input to the cell (and higher membrane potentials), the value of the inactivation parameter for the A-current decreases, so it is less effective in prolonging the hyperpolarization. Thus, it gives the cell the ability to perform a “frequency encoding,” transforming the membrane potential level to spike frequency. This is evident during the bursts shown in Fig. 7.1B. As the B-current grows and increases the average after-spike membrane potential, the firing rate increases. Toward the end of the burst, as the C-current begins to dominate and hyperpolarize the cell, the firing slows. Exercise 4 provides an opportunity to examine this behavior in detail.

7.4 A Model Molluscan Neuron

The cell that is modeled is a “generic burster,” loosely modeled after the *Aplysia* R15 cell at temperatures below 16°C, although it contains channel models taken from measurements on bursting neurons in *Tritonia* and *Anisidoris* as well. Thus, the simulation is presented as an exercise in modeling and a study of the various conductances discussed in the previous section, not as an accurate model of a specific neuron. By implementing the model within the *Neurokit* cell building environment, we have made it possible to modify nearly all of the relevant parameters of the model and its channels while running the simulation. As we have mentioned, the abundant varieties of molluscan pacemaker neurons have numerous features in common. For example, we will see in Exercise 2 that by simply modifying the relative proportions of the six channel types which we have described, we can convert the model from a burster to a beater.

There are some other recent models of the R15 cell that include the channels that dominate the bursting behavior at higher temperatures (Canavier, Clark and Byrne 1991, Canavier, Baxter, Clark and Byrne 1993, Bertram 1993). Butera, Clark, Canavier, Baxter and Byrne (1995) have produced a very detailed model containing 12 state variables and over 50 parameters. More recently, this has been used as the basis for a simplified four-variable model that exhibits many of the characteristics of the activity of the more detailed model (Butera, Clark and Byrne, 1996). After studying Part II of this book, you may wish to use the information given in Chapter 19 to implement one of these models, or to modify the model given here to include these channels.

7.4.1 Adrift in Parameter Space

When choosing the default parameters for a complex model, it would be best if we could obtain all the needed information directly from experimental data, with no adjustable parameters. Unfortunately, this is rarely, if ever, possible.

Chapter 11 discusses some general issues and approaches to guide you in the construction of realistic neural models. In the case of single cell models, the usual approach is to first determine an anatomically correct compartmental representation of the cell and to determine its passive membrane properties. The second step is to incorporate active channels into the model, based on experimental evidence for their existence. As much as possible, the channels should be modeled using the best available experimental data for conductance densities and activation variables. Often, the most complete data come from experiments on different types of neurons in different species, and performed at different temperatures. Also, measurements often show that, even for different examples of the same identifiable cell in the same species, there are large variations of conductance densities and the voltage dependence of channel state parameters.

This uncertainty in the model parameters means that constructing a model neuron usually

involves a third step of searching *parameter space*, in order to find a consistent set of model parameters that will enable the model neuron to duplicate the experimentally observed behavior of the actual cell. This may be quite time consuming, as there may be many parameters that could be relevant to the model's behavior, and there may be large uncertainties in their values. When the channel kinetics are obtained from other neurons than the one being modeled, one can make a first pass at refining the channel parameters by some translation and scaling of the state variables. For example, the voltage dependence of activation curves may be shifted to take into account different resting potentials, or the time constants may be scaled to take temperature differences into account. However, it is often necessary to make a systematic search over parameter space, varying the channel densities and kinetics in order to fit published physiological responses of the cell. The differential equations that describe such a model have a very large number of parameters which could be varied. Although these equations must be solved numerically, there are a number of methods of mathematical analysis that can narrow down the search.

When modeling bursting neurons, it is possible to make use of the vastly different time scales for the spiking and the burst cycle behaviors in order to divide the problem into separate fast and slow subsystems. This can simplify the process of choosing parameters that are consistent with both kinds of behaviors (Rinzel and Lee 1987). Phase plane analysis (Rinzel and Ermentrout 1989) is another useful tool. Instead of plotting one variable as a function of time, one time-dependent variable is plotted as a function of another. For example, the activation variable for the slow hyperpolarizing current may be plotted against calcium concentration, or the membrane potential may be plotted against calcium concentration. Analysis of these plots then suggests ranges of parameters that lead to slow wave oscillations and bursting behavior, and those that lead to beating behavior (Rinzel and Lee 1987, Canavier et al. 1991, 1993). The requirement that plots of the steady-state total channel current as a function of clamp voltage should have a region of negative slope (the NSR) can be used to adjust the maximum conductance for the slow hyperpolarizing current (Bertram 1993).

The next release of GENESIS (likely to be available by the time you read this) will contain a library of routines that may be used to perform automated searches over large regions of parameter space. These may be used to fit the model parameters that govern firing characteristics to the results of current clamp or other experiments (Bhalla and Bower 1993, Vanier and Bower 1996). The parameter search methods implemented in this library include simulated annealing, genetic algorithms, conjugate gradient, stochastic search and simple brute-force search. These are often used to run many versions of a simulation in parallel on a parallel computer, or over a network of workstations (see Sec. 21.7).

For the simple structure of our molluscan neuron, the first step in modeling was relatively straightforward. We have largely neglected the final stage of "fine tuning" our model parameters, leaving this for those readers who are interested in doing so. Instead, we have concentrated on the second stage, and have used the best experimental data that we have

for the channel activations and conductance densities, making changes only as they were required. In our discussion of the implementation of the model, we give the reasons for some of the choices that were made when adjusting parameters. This may give you some guidance when constructing your own models or varying the parameters of this model. The decisions that were made in modeling the hippocampal CA3 pyramidal cell (Sec. 7.6) are described in some detail in the paper by Traub et al. (1991).

7.4.2 Implementation of the Model

One of the first computer simulations of the ionic currents and the resulting action potentials in giant molluscan neurons was performed by Connor and Stevens (1971c), who modeled beating pacemaker neurons in the molluscs *Anisidoris* and *Archidoris*. Like bursting neurons in *Aplysia*, these typically have a single axo-dendritic process extending from a large spherical soma that has a diameter of 250–300 μm . In most of the voltage clamp experiments upon which the channel models were based, the soma was isolated from the rest of the cell. As we are interested in the endogenous firing properties of the neuron, and not in the details of synaptic input, we can model the cell as Connor and Stevens did, treating it as a single soma compartment that contains the voltage-activated channels.

In our simulation, it is convenient to model this with a cylindrical compartment that has both a length and a diameter of 250 μm , and thus has the same surface area as a sphere with a 250 μm diameter. The membrane potential of this compartment will then obey an equation similar to Eq. 2.1, or to the combination of Eqs. 4.1 and 4.3 which describe the behavior of a section of the squid giant axon. As we will have additional channels, we can write it in the form

$$C_m \frac{dV_m}{dt} = \frac{(E_{rest} - V_m)}{R_m} + \sum_k I_k + I_{inject}, \quad (7.1)$$

where the sum over k represents a sum over the different types of ionic currents I_k that are present.

As in Chapter 5, the membrane resistance R_m (in ohms) and capacitance C_m (in farads) are related to the surface area of the cell A and the specific resistance and capacitance by $R_m = R_M/A$ and $C_m = A C_M$. In the model, we have used the Connor and Stevens values of the specific membrane resistance, $R_M = 4.0 \Omega \cdot \text{m}^2$ ($g_{leak} = 0.25 \text{ S/m}^2$), and the specific capacitance, $C_M = 0.07 \text{ F/m}^2$. Likewise, we have used their value for the nominal resting potential of the cell, $E_{rest} = -40 \text{ mV}$. Although an endogenous burster is never “at rest,” this potential is close to the threshold at which action potentials begin in the R15 and other molluscan bursters. The value of the specific membrane capacitance, which is close to the value measured for the *Aplysia* soma (Adams and Gage 1979a), is much larger than the nearly universal value of 0.01 F/m^2 found for most cell membranes. It is likely that this large value arises because the highly infolded membrane of the molluscan soma has a much

Current	p	q	E_{rev} (mV)	\bar{g} (S/m^3)	τ_X (msec)		τ_Y (msec)	
					-40 mV	0 mV	-40 mV	0 mV
Na	3	1	50	138	12	1.6	50	17
K	2	0	-68	66	61	31	—	—
Ca	2	1	64	65	4	5.4	100	177
B	1	0	68	5	1200	240	—	—
C	1	0	-68	124	3800	3800	—	—
A	4	1	68	82	12	12	235	235

Table 7.1 Exponents for activation (p) and inactivation (q), reversal potentials (E_{rev}), conductance densities (\bar{g}) and time constants (τ_X and τ_Y) for the ion channels used in the molluscan burster model. For a given channel type, the conductance per unit membrane area is $\bar{g}X^pY^q$.

larger surface area than that calculated by assuming a smooth spherical surface, as we have done here (Coggeshall 1967). All of these parameters, as well as the soma dimensions, may be varied in the simulation.

For each conductance in the model, we write the ionic current in the form

$$I_k = A\bar{g}X^pY^q(E_{rev} - V_m), \quad (7.2)$$

where A is the area of the soma, \bar{g} is the conductance density (maximum channel conductance per unit area), E_{rev} is the reversal potential for the ion, and the variables X and Y represent its activation and inactivation state variables.

Note that I_k appears on the right-hand side of Eq. 7.1. This, along with the sign of $(E_{rev} - V_m)$ in Eq. 7.2, indicates that we are using the convention that a current of positive ions *into* the cell is to be considered to be a positive current, rather than the “physiologists’ convention” introduced in Chapter 4. We will use this sign convention throughout this chapter and in plotting currents in the two tutorial simulations. For each of the ion currents used in the model, Table 7.1 summarizes the values of the channel conductance densities \bar{g} , the reversal potentials E_{rev} , and the exponents p and q for activation and inactivation. It also gives the time constants for activation and inactivation (τ_X and τ_Y) at E_{rest} (-40 mV) and at a voltage during an action potential (0 mV).

7.4.3 Modeling the Channels

The six types of channel conductances were modeled following the procedures described in Chapter 19. This involved fitting the voltage dependencies of the steady-state activation and inactivation parameters and their time constants to data taken from published results of voltage clamp experiments on molluscan pacemaker neurons. Except as noted, the channel conductance densities given in Table 7.1 have been estimated from these experimental data.

In the model, we have used more significant figures than are justified for most of the conductance densities. We will leave it as an exercise for you to vary their values and to determine which parameters, if any, may be sensitive to changes on the order of 10% or more.

Action Potential Related Conductances

The data for the fast sodium conductance were taken from measurements on the *Aplysia* R15 cell by Adams and Gage (1979b). However, the activation curve X was shifted downward by -5 mV , making it more like that for the fast inward current conductance quantified by Connor and Stevens (1971b). This produced more robust firing when used with the other conductances in the model.

The delayed potassium conductance parameters were fitted to data taken by Thompson (1977) from bursting LPL2 and LPL3 cells in *Tritonia*. However, the potassium reversal potential was taken to be that measured in *Aplysia* by Adams and Gage. Inactivation of this current was not included in the model.

The high threshold calcium conductance was modeled from *Aplysia* R15 cell measurements by Adams and Gage (1979b). The measured reversal potential shown in Table 7.1 is lower than that for most mammalian neurons. This is due to the relatively high concentration of Ca^{2+} ions inside the cell of molluscan neurons (Adams and Gage 1979a).

Burst Related Conductances

The bursting conductance (B-current) was modeled from data taken from bursting cells RPL2, LPL2 and LPL3 in *Tritonia* (Smith and Thompson 1987). However, the current due to this conductance was treated as a pure Ca^{2+} current, rather than that from a combination of Ca^{2+} and Na^+ . This is probably more like the situation in *Aplysia* (Gorman et al. 1982). In order to provide sufficient depolarizing drive to maintain the burst for several seconds, it was necessary to increase the conductance density to a value several times greater than that found in the *Tritonia* measurements. Smith and Thompson noted that the amplitude of I_B in different preparations was quite variable, more so than any other current measured. It was largest in cells that were bursting most strongly. Note from Table 7.1 that this conductance is nevertheless far smaller than that for any of the other conductances. The reversal potential (64 mV) is close to that for the calcium current, as would be expected. However, it was difficult to measure, and was determined only within the very large experimental uncertainty of $\pm 59\text{ mV}$. You may wish to explore the effect of increasing the value E_{rev} and decreasing the value of \bar{g} when experimenting with the simulation.

The steady-state activation for the calcium-activated potassium conductance (Fig. 7.2E) was fitted to the previously discussed *Aplysia* R15 measurements of Gorman and Thomas (1980) when the internal Ca^{2+} concentration was held constant at approximately 187 nM . The value for the conductance density ($\bar{g} = 124\text{ S/m}^3$) was estimated from the Gorman

and Thomas results, and used with no further adjustments. As the activation was found to depend linearly on the Ca^{2+} concentration, we can determine the channel current at any other concentration by scaling Eq. 7.2 by the factor $[Ca^{2+}]/187$, where $[Ca^{2+}]$ is the internal Ca^{2+} concentration in nM . Thus, we need to model the way that Ca^{2+} varies with time and membrane potential. In addition, it is necessary to estimate the time constant for the intrinsic voltage dependence of the channel activation, independent from time variations in $[Ca^{2+}]$.

Chapter 19 describes how the variation of $[Ca^{2+}]$ may be modeled by an equation of the form

$$\frac{d[Ca^{2+}]}{dt} = B I_{Ca} - \frac{[Ca^{2+}]}{\tau}. \quad (7.3)$$

The first term on the right gives the rate of increase of $[Ca^{2+}]$ due to an inward channel current I_{Ca} . In this model, the current is the sum of the B-current and the high threshold calcium current. As discussed in Sec. 19.4.1, it is difficult to calculate the constant of proportionality B from first principles. A value of 1000 moles/ampere/ m^3 was chosen in order to give reasonable bursting behavior and increases of $[Ca^{2+}]$ that are typical of those measured during a burst. The second term represents an attempt to fit the various processes that deplete $[Ca^{2+}]$ to an exponential decay with a single time constant, τ . Gorman and Thomas found τ to be 17.5 sec from their Ca^{2+} injection experiments.

The remaining question is to determine the time constant for activation of the voltage-dependent conductance for a fixed $[Ca^{2+}]$. Gorman and Thomas estimated that for a depolarizing step from -50 mV to 110 mV , the time constant for activation is only 12 msec. However, this is far outside of the region of interest for our simulation, as we are particularly interested in the rate of decay of the hyperpolarized phase between bursts. It is also fairly common for activation time constants to be much larger at low voltages than at higher voltages. Thus, we have estimated the time constants from measurements made by Thompson et al. (1986). They found that when *Aplysia* L2 cells were repolarized to -40 mV after a depolarizing step, the C-current decayed exponentially with a time constant of about 3.1 sec. As this includes the effect of the decay of $[Ca^{2+}]$, we estimated the activation time constant τ_X from the relation $1/\tau_X + 1/\tau = 1/\tau_{net}$, obtaining the value of 3.8 sec listed in Table 7.1. For convenience, it was assumed to be constant over the range of interest.

The transient potassium conductance model was taken from the Connor and Stevens (1971c) measurements of the A-current activation and inactivation state variables in beating *Anisidoris* pacemaker neurons. These are very similar to those found by Thompson (1977) for bursting *Tritonia* neurons. For consistency with the other potassium conductances used in the model, a reversal potential of -68 mV was used, rather than the slightly higher value measured in *Anisidoris*.

7.5 The Molluscan Neuron Simulation

The model that we have described in the previous section has been incorporated into a simulation based upon the *Neurokit* cell builder environment. This tutorial then provides an introduction to *Neurokit*, as well as an opportunity to further explore the roles that these six conductances play in determining neural behavior. *Neurokit* is an elaborate GENESIS simulation that can create a cell model from a file which describes the morphology of the cell and the channels which it contains. Once a cell has been loaded into *Neurokit*, you will be able to modify it extensively and perform a variety of experiments. This means that you will be able to use the same simulation and menu system to study the properties of both the molluscan burster and the more complex CA3 pyramidal cell model.

Neurokit may be used with multi-compartmental models to change the geometry and electrical characteristics of compartments, and to add, delete, and reposition them within the neuron. It also allows channels to be deleted from compartments and for new channels to be added from one of the GENESIS libraries of predefined channels. Its channel editing facilities may be used to modify activation curves and their time constants. Thus, it is possible to construct and modify sophisticated single neuron models with little or no GENESIS programming.

In this tutorial, we explain and use only a few of the basic features of *Neurokit*. The experiment described here and the exercises at the end of the chapter should help you to learn more about the ways that the six basic channel types of this model can affect its firing patterns. If you wish to learn more about the advanced features of *Neurokit*, you may run it from the *Scripts/neurokit* directory and try some of the examples that are suggested in the on-line help. (Alternatively, you may print out the *README* file.) In Part II of this book, Chapter 17 describes how you may use *Neurokit* to implement your own cell models.

7.5.1 Using Neurokit

In order to allow room for the graphical display, move a terminal window into the lower left-hand corner of your screen. (This is normally done by using the left mouse button to click and drag on the title bar of the window.) After changing into the directory in which the simulation resides (usually *Scripts/burster*), type “genesis Neurokit” to the UNIX prompt. Chapter 17 explains the details of the initialization files that are used by *Neurokit* to load a particular cell model. There will be a delay while the simulation is being set up, and eventually, a title bar with the options

```
quit    help    file    run cell    edit cell    edit compt    edit channel
```

will appear at the top of the screen. Click on the *help* button, and a brief summary of the model and instructions for running the simulation will be displayed. When you feel that you

are ready to begin, click on the CANCEL button at the bottom of the help window. The next time that you call up help, you will return to the same place in the help file.

Click on the **file** option on the title bar. The **file** menu will appear, showing various dialog boxes which indicate that we are running the **mollusc** cell model simulation. As the default values in the boxes are all correct, click the **Load from file** button on the **file** menu. A few obscure messages will appear in your terminal window. Then click **run cell** on the title bar. You should see the **SIMULATION CONTROL PANEL** and two “Cell Windows,” each containing a representation of the cell with a graph below. The graphs have been initialized to plot the membrane potential at the left and the internal concentration of calcium ions at the right. (In our SI units, 10^{-6} moles/m³ = 1 nM concentration.) In other experiments, we may use the **scale** button in the upper left corner of the second Cell Window to plot other quantities of interest, such as the current, conductance, or activation variable for a given channel.

Initially, the **Recording** button under the **ELECTROPHYSIOLOGY** heading will be highlighted, indicating that you are ready to plant recording electrodes in the cell. Start by clicking the left mouse button in the round soma in each Cell Window. A recording electrode should appear in each cell and the boxes labeled **click_site1** and **click_site2** should each display “/mollusc/soma”. Other buttons under the **ELECTROPHYSIOLOGY** heading provide options for performing current clamp and voltage clamp experiments and for applying various types of synaptic input to models that contain synaptically activated channels.

We would like to begin by examining the endogenous bursting of this model in the absence of any input, so go ahead and click on **run** in the **SIMULATION CONTROL PANEL**. The dialog box for **runtim e** indicates that the simulation will run for 40 seconds of simulated time each time you click on **run**. The **refresh_factor** dialog indicates that the graph will be updated every five time steps. This gives a reasonable compromise between execution speed and accuracy in depicting the details of the action potentials, but you will notice that the action potentials will have irregular heights. If you need to examine individual action potentials in detail and don’t mind the slower performance, you may wish to set this value to “1” and decrease the time step in the **clock** dialog box. If you wish to examine a burst in more detail, or to run the simulation for a longer period of time, click on the **scale** button in a graph window to bring up a menu for changing the scales used for the plots.

7.5.2 Understanding the Results

Once the run has finished, let’s try to understand what we see in the two plots. The simulation begins with all channel conductances initialized to their steady-state values when the membrane potential is clamped to the nominal resting potential of -40 mV . Of course, this cell is never “at rest,” so the initial burst of spikes will be slightly different from the following ones. After the end of the first burst of action potentials, there is a gradual decrease in V_m , reaching a maximum hyperpolarization of about -58 mV . The membrane potential then

begins to increase. Once it gets over about -50 mV , the rate of increase accelerates. Once it gets near -40 mV , the cell begins firing again. At this point, the calcium concentration is at its minimum value. During the burst of action potentials, the concentration rises rapidly, reaching its maximum at the end of the burst. It then begins a slow decline during the interburst interval.

We can understand this behavior by looking at the activation variables and their time constants. The activation variables are plotted in Fig. 7.2, and Table 7.1 shows the corresponding time constants at membrane potentials of -40 mV and 0 mV . However, *Neurokit* allows us to both edit and view the voltage dependence of the steady-state activation variables, their time constants, and the corresponding Hodgkin–Huxley rate constants α and β (Eqs. 4.21 and 4.22). This may be done by clicking on `edit channel` in the title bar. This will bring up a Cell Window at the lower left containing the green spherical soma. Click on the soma. It will turn red, and icons representing the soma and the six channel types will appear in the Compartment Window at the upper right. Begin by clicking on the icon for the B-current in order to select it. The `gate` dialog box will show the default gate to be examined, the X gate. Position the cursor in this box and hit “Return.” The resulting plots will reveal that the slow inward B-current begins activating at about -50 mV , with a time constant of about 2 seconds. (The graph for steady-state activation or inactivation curve plots is labeled `m_inf` to correspond to the symbol m_∞ often used for the Na activation gate.) This should explain the basis for the buildup of the depolarization that allows action potentials to be generated during the burst. Also examine the activation and inactivation (Y gate) for the Ca current. When is it likely to make a significant contribution to the Ca^{2+} concentration? Examine the activation of the C-current. In the simulation menus and icons, we represent this type of conductance by the symbol `K_C`, in order to distinguish it from the faster potassium conductance `K` which is associated with action potentials. The outward current due to this channel will be proportional to its activation, the internal Ca^{2+} concentration, and the difference between the membrane potential and the Ca reversal potential. Can you use this to understand how the burst is terminated and how the hyperpolarized interval comes to an end? Later, you may wish to experiment with the effects of applying offsets (`ox` and `oy` dialogs) or scaling (`sx` and `sy`) of the activation curves or their time constants. This procedure is described in Exercise 6 and is discussed in more detail later in Part II, Sec. 19.2.2.

It may also be helpful to examine the various channel currents over the course of the simulation. We could use the second Cell Window to plot one of the channel currents instead of the Ca^{2+} concentration, but there is another way that will let us see all six channel currents at the same time. The `run paradigm` button in the SIMULATION CONTROL PANEL may be used to execute some previously defined function that is not normally part of *Neurokit*. In this particular simulation, clicking on `run paradigm` calls up a window containing plots for the six channel currents. Examining these currents before, during, and after a burst should give you further insight into the interplay between the various conductances during

this cycle. Note that we have used the GENESIS convention of plotting inward currents as positive and outward currents as negative. The Hide Plots button at the top left of the Channel Currents window may be used to temporarily hide the plots from view while the plotting continues. These additional plots will slow the simulation down considerably. The Delete Plots button at the top right of the window will disable plotting of channel currents completely.

Other experiments given in the exercises at the end of the chapter will let you explore the effects of the various ionic conductances on the firing patterns of the model neuron.

7.6 The Traub Model CA3 Pyramidal Cell

We conclude this chapter with a tutorial based on a more complex mammalian neuron model that incorporates channels very similar to the molluscan channels we have described. The associated tutorial simulation uses a *Neurokit* implementation of the hippocampal CA3 region pyramidal cell model of Traub et al. (1991). This single cell model was developed for use in network models, in order to study the synchronized burst firing of CA3 pyramidal cells that occurs during epileptic seizures. It is often the case in computational neuroscience that one begins with an interest in studying the “high level” network behavior of a system, but finds that it is necessary to first model the detailed behavior of the “low level” components in order to reproduce the behavior of the system.

The implementation of this model is described in considerable detail in the original paper, and our treatment is very brief. Thus, it is recommended that this paper be consulted while using the tutorial. In the tutorial, we will explore some of the properties of this model, as well as the ways that these basic conductance types may interact within a multi-compartmental model. The simulation and the details of the model closely follow the description in the paper, except that: (a) Unless otherwise noted, SI units are used throughout. (b) The membrane potential V_m is measured relative to an extracellular potential of zero; thus the nominal resting potential is -0.06 V. (c) The synaptically activated channels in compartments 3 and 15 have not been implemented. A more recent 64-compartment model with dendritic branching (Traub, Jeffereys, Miles, Whittington and Tóth 1994) may be found in the cell model archives of the GENESIS Users Group. (See Appendix A.)

This simplified model of the CA3 pyramidal cell has 19 compartments with a soma in the center and two linear chains of compartments to represent the apical dendrites and the basal dendrites. As does the molluscan burster, this cell has fast sodium, delayed rectifier potassium, high threshold calcium, transient potassium, and calcium-activated potassium conductances. As one might expect for neurons in warm-blooded animals, their time constants for activation and inactivation are much smaller than those observed in molluscs. However, the burst is maintained, not by a slow inward B-current, but by an interesting interaction of events in the soma and the dendrites.

Calcium channels turn out to be particularly important in determining the behavior of this cell. Calcium imaging studies (Regehr, Connor and Tank 1989) have shown that, unlike Na channels which are concentrated in the soma, Ca channels are widely spread over the cell membrane, with high concentrations in the distal dendrites where no Na channels are found. Calcium has a high reversal potential, so they can produce action potentials like sodium. However, these high threshold Ca channels become activated at higher voltages than the Na channels, so their conductance needs to be triggered by sodium action potentials. They also have a much larger time constant for inactivation—on the order of 200 msec. The result of this concentration of high threshold calcium channels in the dendrites is that broad action potentials may be triggered in the dendrites by passive propagation of soma action potentials.

Not only do Ca channels have a direct effect on the membrane potential, but they have an indirect effect as well. These slow calcium spikes in the dendrites propagate back to the soma to depolarize it and sustain the burst. As with the molluscan burster, the Ca^{2+} entry during the burst activates potassium channels that conduct outward hyperpolarizing currents, quenching the burst. In this case, the outward current is a sum of two currents. The voltage- and calcium-dependent potassium C-current conductance is similar to that found in molluscs, but has a relatively fast response to changes in the membrane potential at the higher ambient temperatures in mammals. These cells and other mammalian neurons also contain another current that depends only on the much slower variations in Ca^{2+} concentration. This is the AHP (*after-hyperpolarization*) current, which is responsible for the period of hyperpolarization after a burst of action potentials. The transient potassium A-current plays a relatively minor role in this model. It affects the interval and onset of action potentials.

Published data from voltage clamp experiments were used to model the dynamics of these channels and to make reasonable hypotheses about their distribution throughout the cell. Once we have verified that the model can reproduce experimental results, we can use the simulation to understand the ways in which the various channels interact and produce distinctive bursting patterns of action potentials.

7.6.1 Experiments with the Traub Model

Let's begin by trying to reproduce the bursting behavior observed in this cell and described in the paper by Traub et al. Change to the *Scripts/traub91* directory and begin the simulation in the same manner as the *burster* tutorial, starting GENESIS first, and then typing “Neurokit”, once GENESIS has begun. As before, use the file menu to load the default cell model. In this case, it will be “CA3.” After clicking on `run cell` in the title bar, you will be ready to try some experiments.

We would like to record from the soma and from a compartment (apical_14) in the apical dendrite which is 0.5 length constants from the soma. These two compartments are of interest

because of their very different concentrations of ion channels. The soma contains the highest concentration of Na channels. Compartment apical_14 and its adjacent compartments in the apical dendrite contain the highest concentration of Ca channels, but have no Na channels to produce sharp action potentials.

Start by clicking the left button in the soma. A recording electrode should appear and the box labeled `click_site1` should display “/CA3/soma.” To plant a recording electrode in compartment 14 of the apical dendrite, click the left mouse button about half way to the right along the “fat” dendrite extending from the soma. If the `click_site1` box then shows something other than “/CA3/apical_14,” click the middle button in the same place to remove the electrode and try again. With a little practice, you will find it easy to record from any desired part of the cell.

Next, click on `Iclamp` under the ELECTROPHYSIOLOGY heading. Underneath, a dialog box labeled “`inject (nanoAmps)`” should appear, with a value of “0” in the box. Set the injection value to 0.2 nA . Then click on the soma to plant an injection electrode in the soma. Finally, click on `run` in the SIMULATION CONTROL PANEL to start the simulation. Note that most of the buttons and dialog boxes may be used while the simulation is running. For example, you may call up the help window or change the injection current without stopping the simulation. When running long simulations, you may pass the time by consulting the help window or by inspecting the model parameters with the `edit cell` option (described below).

While the simulation is running, you will see the compartments in the cell diagram change color to represent the value of the membrane potential V_m . “Hot” colors represent higher voltages, and “cold” colors represent lower voltages. With some practice, you may use this feature to observe the propagation of action potentials along the dendrites. The `runtime` dialog box shows that the simulation will run for 0.1 seconds of simulation time, and the `current_time` box displays the current simulation time. `clock` shows the default integration time step of “ $5\text{e-}5$ ” ($50\text{ }\mu\text{sec}$). The `refresh factor` of “10” indicates that the graph and the cell diagram will be updated every 10 simulation steps. You may speed up the simulation by increasing this number, at the expense of losing resolution in the plot of action potentials.

The plots below the cell diagram show the membrane potential in the two compartments. The lower one shows a burst of action potentials in the soma that terminates after about 50 msec . If we were to run the simulation longer, we would see another burst in about a second. The curve above, which has been displaced by 100 mV for clarity, shows the membrane potential in the middle of the apical dendrite. Here the spike is very broad and somewhat delayed. There are no Na channels in this region, but the density of Ca channels is high, so we might suspect that we are seeing a Ca action potential on top of whatever soma potential has propagated to this point. This point is 0.5 length constants from the soma. You should verify that the theory of Chapter 5 predicts that the action potentials from the soma will be attenuated by a factor of about 0.6.

We can understand the interaction between these two types of action potentials by paying close attention to the colors in the cell diagram as we step the simulation for short amounts of time. To slow things down a bit, set the `clock` dialog box value to “`1e-5`” seconds. Then set the `runtime` dialog value to “`0.016`”, so that the simulation will stop just before the beginning of the burst. Click on the `reset` button in order to reset the simulation time to zero and clear the graph. Then click on `run` again to perform 16 *msec* of simulation. Now change the `runtime` to only 0.001 seconds, and click on `run` again *without* clicking on `reset`. You may step through the simulation at 1 *msec* intervals in this manner, watching the propagation of action potentials along the apical dendrite. You should be able to see how the Na action potentials that are generated in the soma propagate to the dendritic compartments where they trigger Ca action potentials. These then propagate back to the soma where they cause the depolarization that maintains the burst and increases the spike frequency during the middle of the burst. Figure 7.3 gives a gray-scale “snapshot” of the situation near the peak in the dendritic calcium spike.

It is also possible to plot other quantities such as channel current or channel conductance on a second graph. In the *burster* tutorial, this graph appeared by default to plot the Ca^{2+} concentration in the soma. Looking at these other variables can help us to understand what mechanisms are responsible for terminating the burst. Click on `Show extra cell window` to bring up a second cell diagram and its associated graph. If you click on the `scale` button in the upper left corner of the second Cell Window, a menu will appear containing a number of dialog boxes. The only two that are relevant at present are `colfield` (containing “`Gk`”) and `fieldpath` (containing “`Ca`”). The first of these is the name of the quantity to be plotted on the graph and to be used to determine the color of the compartments. In this case, it is the channel conductance “`Gk`”. The `fieldpath` contents specify the location of the field to be plotted, i.e., the Ca channel. The compartment is specified in the usual way by planting recording electrodes in the cell diagram. Other channels that you can choose are the Na, K_DR (delayed rectifier), K_AHP (after-hyperpolarization), K_C (C-current), or K_A (A-current) channels. For `colfield` you may choose the channel current (“`Ik`”), activation variable (“`X`”), or inactivation variable (“`Y`”), as well as the conductance (“`Gk`”). In addition to the channels, each compartment has an element called “`Ca_conc`” which keeps track of the Ca^{2+} concentration. In order to plot the concentration, you may set `fieldpath` to “`Ca_conc`” and `colfield` to “`Ca`.” The `scale` button on the associated graph should be used to set the graph scales to a range consistent with the quantity to be plotted.

For now, let’s go with the default values and plot the Ca channel conductance. Click on `APPLY_AND_VANISH` to put the menu away. Click on the `Recording` button under the `ELECTROPHYSIOLOGY` heading and plant electrodes in the soma and apical.14 compartment of the second Cell Window. (It is not necessary to apply injection in the second Cell Window, as the injection is being applied in the first.) Set `clock` back to “`5e-5`,” `runtime` to “`0.1`,” then click on `reset` and `run`. Note that multiple plots in this second graph window do not have a vertical offset, as is the case with the V_m plots in the first graph. You should

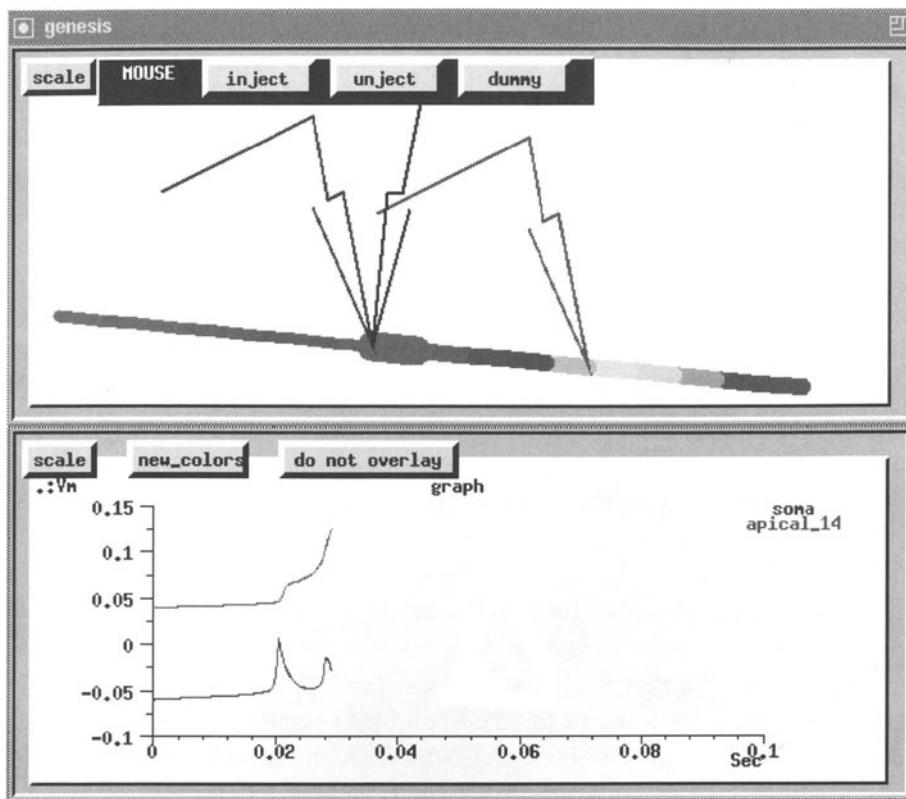


Figure 7.3 The cell diagram for the *traub91* tutorial, showing recording and injection electrodes, and the propagation of action potentials. The plots below show the membrane potential in the soma (lower trace) and in apical dendrite compartment “apical_14” (upper trace).

see a large broad peak in the Ca conductance in the dendrite, corresponding to the voltage peak.

We can make use of the “overlay” feature of the graph in order to compare the timing of the Ca conductance with that of the K_C conductance. In order to overlay the results of a second run using new colors, click on new_colors and do not overlay at the top of the second (conductance) graph. The label should then change to “overlay,” indicating that the graph will not be cleared on reset. Click on the scale button of the second Cell Window, change the fieldpath to “K_C,” and then click on APPLY_AND_VANISH. Finally, click on reset, and then run in the control panel. Notice the position of the peak in the K_C conductance in the apical_14 compartment relative to that of the peak of the Ca conductance. How do these peaks correlate with the shape of the action potential in this compartment?

As these results indicate that Ca and calcium concentration-dependent K conductances

have a lot to do with the bursting behavior, it would be reasonable to repeat the experiment with the Ca channels blocked in order to see if this destroys the bursting behavior. We can use the `edit_cell` selection on the title bar to set the Ca conductance density to zero in every compartment. After you click on this button, a Cell Window with a drawing of the cell should appear in the lower left corner of the screen. Click on the left end of the diagram. The selected compartment will turn red and a labeled diagram of the compartment will appear in the Compartment Window at the upper right of the screen. In this case, the compartment will be the basal_1 compartment. As it contains no channels, click on the next compartment in the Cell Window. The basal_2 compartment should now appear in the compartment window, accompanied by icons representing the Ca, K_AHP, and K_C channels. If you click on the Ca channel icon, the `Selected channel` dialog box will show “/CA3/basal_2/Ca,” and the Conductance dialog box will show the default conductance density (50 S/m^2). Set this value to zero. Now click on the next compartment in the Cell Window. You should now be inspecting the Ca channel of the basal_3 compartment. (As the Ca channel is the currently selected channel, you will not need to repeat the channel selection.) Set the conductance density to zero and go on to the next compartment. When the Ca conductance density has been set to zero in all the compartments, select `run cell` from the title bar. (HINT: It is also possible to set various parameters by typing commands to the “genesis” prompt. The command “`setfield /CA3/#/Ca Gbar 0.0`” uses *wildcard* symbols (#) to set the maximum Ca conductance to zero in all the compartments. Chapter 12 explains the use of the GENESIS `setfield` command.) Now click on `reset` and repeat the sequence of steps for plotting the results of applying a 0.2 nA injection current to the soma. Can you qualitatively explain the different interval between action potentials in the absence of Ca currents?

7.6.2 Firing Patterns

Experiments have shown that CA3 pyramidal cells have different modes of firing with different amounts of somatic current injection. For small depolarizations, the pattern is one of short bursts separated by a fairly long hyperpolarized interval which decreases as the amount of injection increases. With larger injections, there is a transition to repetitive firing of single action potentials. This can be observed by performing 1.5 second runs with currents of 0.2, 0.3 and 0.5 nA injected to the soma.

After having performed the previous experiment, the fastest way to restore the original conductances and simulation parameters is to quit the simulation and reload the CA3 cell model. Then, proceed as in the first experiment, with a single Cell Window and graph. Record from the soma and the apical_14 compartment and inject a 0.2 nA current to the soma. However, the `runtimes` value should be increased to 1.5. The default time scale for the graphs is only 0.1 second, but the scales may be changed while the simulation is running.

Run the simulation and, once the simulation has run past 0.1 second, click on the **scale** button associated with the graph. Change the value in the **xmax** dialog to “1.5” and press the **APPLY_AND_VANISH** button. You may also adjust the value of **xmin** in order to zoom in on a particular pattern of bursts or other features. Note that the existing data will be replotted whenever the scale is changed. Use **overlay** mode and new colors to superimpose plots of runs with injections of 0.3 and 0.5 nA . Note the transition between the two firing modes. If you have the time, you might also try injections of 0.1 and 0.8 nA . If you get bored during these long runs, you may amuse yourself by looking at the various options and information available through the **edit_cell**, **edit_compt** and **edit_channel** title bar selections. The simulation will continue to run in the background.

You should note that, with the larger amounts of injection, the cell will settle down to a pattern of regular firing after an initial burst. The larger depolarization provided at these higher injection currents is strong enough to cause somatic action potentials, in spite of the shunting of the membrane by the AHP and other outward currents. However, this shunting is enough to severely attenuate the action potentials by the time they reach the dendrites. Although they cause increases in the Ca conductance, they are unable to provoke full-blown Ca action potentials in the dendrites. The result is that enough Ca enters the cell to maintain the dendritic shunt from the calcium-activated conductances, but not enough to stop the production of action potentials in the soma. Exercise 8 asks you to further explore the details of this firing mode and of a third mode that is predicted by the model.

7.7 Exercises

1. Some explanations or models of bursting in the *Aplysia* R15 neuron ignore the high threshold Ca current. Explore the effects of eliminating this current by setting its maximum conductance to zero. You may do this by selecting **edit cell** from the top menu bar. Click on the soma in the lower left window and then click on the icon for the Ca channel in the upper right window. Change the Conductance dialog value from 65.2 to 0, and hit “Return.” Click on **run cell** in the title bar. Finally, click on **reset** and **run**. Explain the reasons for the differences that you see.
2. The *Aplysia* L11 cell is a repetitively discharging pacemaker (“beater”) instead of a “burster.” Gorman and Hermann (1982) have estimated that in the R15 cell, the conductance density for the B-current channels is 6 times greater, that of the high threshold Ca channels is 8 times greater, and that of the K_C channels is 23 times greater than in the L11. Use **edit cell**, as described in the previous exercise, to transform the model cell to a “beater.”
3. When sodium channels are blocked with TTX, a slow oscillation in the R15 membrane potential is seen. Set the Na conductance to zero and compare the period of the

resulting slow oscillation to that of the bursts when the Na conductance is present. How does it compare to the period of the bursts in Exercise 1? Can you explain any differences? (For these measurements, use the **scale** button on the graphs to increase the time scale to 80 seconds and run the simulation for 80 seconds.)

4. Examine the effect of the A-current by performing one of these two experiments on the molluscan burster model.
 - (a) Use the default parameters for producing bursts and set the time scale for the V_m plot so that you can observe the second burst in detail. Notice that the firing rate increases at the beginning of the burst and then decreases at the end, approximately following the pattern of the “slow oscillation”. After studying the results, set the channel conductance for the A-current to zero and repeat the experiment. What differences do you see? Also compare the shape of the action potentials with and without the A-current. Explain these results in terms of what you know about the properties of the A-current. It will be useful to refer to the A-current activation/inactivation curves and their time constants.
 - (b) Alternatively, examine the effects of the A-current in the absence of the Ca-, B- and C-currents by setting their maximum channel conductances to zero. This will leave only the Na-, K- and A-currents present. Without the inward B-current, you will need some current injection in order to produce firing, so set the injection current to 1 nA. (You may do this by clicking on **Iclamp**, planting an injection electrode in the soma in the left Cell Window, and entering the current in the **inject** dialog box.) Estimate the firing rate and repeat for 2, 3, 4 and 5 nA injections. Then remove the A-current and estimate the firing rate for these injection currents. As in part (a), explain the differences in response and in the shape of the action potentials.
5. In Exercise 8 of Chapter 4, we used the *Squid* tutorial to examine the phenomenon of “anode break” (posthyperpolarization rebound). Insert a current injection electrode into the molluscan cell as described above, but leave the injection current set to zero. Run the simulation for about 15 seconds, so that it ends during the interburst interval. Now set the injection current to -5 nA and the run time to 5 seconds. After running for 5 seconds, restore the injection current to zero and run for another 15 seconds. Explain the reasons for the differences between this experiment and the case with “squid-like” Hodgkin–Huxley channels. (Hint: it may be useful to observe the sodium inactivation variable during the course of the experiment. You may plot it, rather than the Ca^{2+} concentration, by clicking on the **scale** button in the right-hand Cell Window (labeled “molluscxout2”). When the scale menu comes up, change the **colfield** dialog box entry to “Y” and the **fieldpath** to “Na.” Now use the **scale** button on the graph below to set **y_{max}** to 1.)

6. In Sec. 7.3, we mentioned that some molluscan pacemakers have a “shoulder” on the downward slope of their action potentials (Aldrich, Getting and Thompson 1979b). This is not seen in recordings from *Aplysia* R15 cells, nor in our simulation. Examine the effect of the Ca conductance on the width and shape of the action potentials in the molluscan burster model. It will be best to “zoom in” on the second burst by adjusting the values of `xmin` and `xmax` in the `scale` menu for the graph that plots V_m . Although it will slow the simulation down somewhat, you may also wish to set the `refresh_factor` dialog value to a lower value in order to increase the resolution of the plot.

After using `edit cell` to remove the Ca conductance and noting whether or not this has any significant effect on the action potentials, restore the conductance and try shifting the threshold for Ca activation to lower voltages. This may be done with the `edit channel` selection. Follow the procedure given in Sec. 7.5 for examining the Ca channel activation gate. Then apply a negative offset to the voltage axis for the gate by entering it in the `ox` dialog box. You might begin with a value of “`-0.010`” in order to shift the activation curve down by 10 mV. Clicking on the `m_inf` button will apply this offset to the steady-state activation curve (m_∞ , which we have called X_∞). Each time you click on `m_inf`, the offset will be applied to shift the curve down by an additional 10 mV. Examine the second burst with various thresholds for Ca activation. Describe and explain your results.

7. In an earlier experiment with the CA3 pyramidal cell model, we studied the behavior of the Ca and K_C conductances during a burst. In a similar manner, look at the K_AHP conductance during the interval between bursts. What happens when this conductance is set to zero in all compartments? What happens when the K_AHP conductances are restored and the K_C conductances are removed instead? Use these results to explain why, unlike the molluscan bursters, the CA3 pyramidal cell requires both of these calcium-activated potassium currents for burst firing.
8. The paper by Traub et al. (1991) describes a third firing mode which is predicted by the pyramidal cell model when the cell is subject to large amounts of depolarizing dendritic stimulation. In the simulation, this can be provided by injecting a moderately large current into the distal dendrite region. Try injecting 1.5 nA into the apical_15 compartment and note the firing patterns. In addition to monitoring both V_m and the Ca conductance, do some runs with plots of the K_AHP conductance for this situation and for the low and high current injections to the soma which were described earlier. Use these results to give a detailed explanation of how the three different firing patterns arise.
9. Try some voltage clamp experiments on the CA3 pyramidal cell model by selecting `Vclamp` from the `run cell` menu. (Before leaving `Iclamp` mode, be sure to use

the middle mouse button to remove any current injection electrodes!) For example, clamp the membrane potential to the nominal resting potential, -0.060 V . Look at the steady-state values of each of the ion channel currents in the soma and other selected compartments. What can one learn from this?

10. Traub et al. (1991) describe two other cell models: an acutely isolated cell (soma plus the two adjacent compartments) and a CA1 region cell model. To load the former model, select the `file` menu from the title bar after quitting and starting a new session with *Neurokit*. Then enter “/acute” in the `Cell for IO` dialog box and “acute.p” in the source file name dialog box before clicking on `Load from file`. As before, select `run cell` from the title bar. When the cell diagram appears, click on `Recording` and plant some electrodes. If the cell diagram is not properly centered, you may move it to the left or the right by hitting “Ctrl-H” or “Ctrl-L” while the cursor is in the Cell Window. Switch to `Iclamp` mode and try some of the experiments described in the paper. Try to reproduce and explain the results shown in Fig. 4 of their paper.

For the CA1 model, the entries are “/CA1” and “CA1.p.” Note that the different distribution of channels yields different firing patterns from those observed in the CA3 model. Why is there no burst firing when injection is provided to the soma?

Chapter 8

Central Pattern Generators

SHARON CROOK and AVIS COHEN

8.1 Introduction

Many organisms exhibit repetitive or oscillatory patterns of muscle activity that produce rhythmic movements such as locomotion, breathing, chewing and scratching. Examples include the escape swimming of the mollusc *Tritonia diomedia*, the digestive rhythms of the lobster, the undulatory swimming movements of the fish or the lamprey, the stepping movements of the cockroach, the rapid wing motion of the locust during flight, and the more complicated locomotion of a quadruped mammal such as the domestic cat. The neuronal circuits that give rise to the patterns of muscle contractions which produce these movements are referred to as *central pattern generators*, or CPGs. Various experimental preparations in which the CPG is isolated from external influence demonstrate that these circuits require no external control for the generation of temporal sequences of rhythmic activity. However, these animals move through the world in an adaptive manner where the same motoneurons are involved in the production of a variety of rhythmic behaviors. Thus, many CPGs are capable of producing multiple patterns of activity in the intact behaving animal (Getting 1989). The ability to switch between different motor behaviors and blend different rhythms relies on feedback from proprioceptors and influence from higher centers of the nervous system; therefore, it is most appropriate to view every CPG as one piece of a distributed control system (Cohen 1992).

One would like to understand how the neurons in a CPG interact and influence one another, how the underlying circuitry of the network produces the collective behavior of the cells, what mechanisms might allow the network to switch among various patterns of

activity, and whether the oscillatory patterns are due primarily to the activity of individual intrinsically oscillatory neurons or to oscillations that are a product of the entire network. The number of cells composing a network that functions as a CPG often determines the manner in which the CPG is studied and the choice of a modeling strategy. Some CPG circuits are anatomically localized and contain a small number of neurons. This occurs most often in CPGs that produce rhythmic behaviors in invertebrates. In these small networks, neurons can be individually identified from animal to animal, permitting detailed circuit descriptions that include cellular and synaptic properties. In contrast to these invertebrate CPGs, there are possibly millions of neurons involved in the production of rhythmic patterns of motor activity in most vertebrates (Murray 1989). In this case, modelers often categorize the neurons into classes that share similar properties so that large networks can be simulated by relatively few cell types (Getting 1989).

The small localized CPGs that occur in invertebrate preparations make it possible to study the relationship between the emergent collective behavior of the biological network and the network's underlying circuitry (Getting 1989). The dynamical properties of many invertebrate CPGs have been analyzed using such techniques as experimental manipulations of cellular, synaptic, and connectivity properties, detailed simulations of the cell interactions within the network, and analytical studies of equations that might describe the network dynamics. For example, Getting created a network simulation of the escape swimming rhythm of the mollusc *Tritonia diomedea* (Getting 1989). This simulation relies on a compartmental model of the network cells with appropriate passive membrane properties, repetitive firing characteristics, and synaptic actions. In addition, the input to the model corresponds to the normal sensory activation of the actual CPG. Some of the properties of Getting's model are demonstrated in the GENESIS simulation *Tritonia*. Another example of an invertebrate CPG that has been studied and modeled extensively is the lobster stomatogastric ganglion. This region contains the neurons that are involved in the generation of the slow rhythm that fires the muscle contractions of the lobster gastric mill and also those that generate the rhythm that controls the muscles of the pyloric region of the lobster stomach (Shepherd 1994). Experimentation with this system has shown that even when a detailed study of the network circuitry provides a qualitative description accounting for the presence of a given motor pattern, there is often no precise explanation for the mechanisms that control the frequency, duration, and phase relations of the motor pattern (Marder and Meyrand 1989). Studies of the invertebrate CPGs mentioned above show that the generation of these rhythms is a complicated process involving the influence of multiple neurotransmitters and modulators that modify the output of the circuit (Marder and Meyrand 1989).

Due to the large number of neurons present in most vertebrate CPG circuits, models of CPGs in vertebrates often involve simplified mathematical representations where a single oscillator may represent many neurons. For example, most models of mammalian locomotion attempt to create an oscillatory network that can account for the production of the alternating flexor-extensor activity responsible for limb coordination during locomotion (Grillner

1981). In such models, the step cycle of each single limb is represented by the cyclic behavior of an oscillator intended to abstractly represent the collective output of the neurons controlling that limb. Examples of such models include Szekely's model for the locomotion of the salamander (Szekely 1968), and the models of Lundberg and Phillips (1973) and Grillner (1981) for cat locomotion. The lamprey and various types of fish have also been used extensively in studies of vertebrate CPGs. These organisms propel themselves through water by a sequence of rhythmic body undulations caused by traveling waves of contractions that progress down the axial muscles from head to tail. Their swimming patterns have led to models of locomotion consisting of chains of coupled oscillators that represent the state of the oscillatory cells that occur in segments along the spinal cord and control the sequence of muscle contractions along the body during locomotion (Rand, Cohen and Holmes 1988, Cohen and Kiemel 1993, Kopell and Ermentrout 1986). For an overview of such models, see Murray (1989) or Cohen, Ermentrout, Kiemel, Kopell, Sigvardt and Williams (1992).

We must understand the mathematical basis for a model if we hope to use it to gain insight into the behavior of the biological system that it represents. In this chapter, we examine some of the fundamental ideas used in the formulation of models of central pattern generators. These mathematical concepts are reinforced with examples in the form of simulations that can be run using the GENESIS script *CPG*. This script creates a simulation environment consisting of a network of four neurons where the user may interactively change the cell parameters as well as the connections between cells and the synaptic properties of those connections. Before continuing with this chapter, the reader should become familiar with the *CPG* tutorial. Begin by entering the *CPG* simulation environment using the process for running a GENESIS simulation which is described earlier in this book in Sec. 3.3. After entering the simulation environment, click on the HELP button located in the control panel at the top center of the screen. The help menu that appears provides some initial instructions under the headings USING HELP, RUNNING THE SIMULATION, and MODEL PARAMETERS. These help texts provide the procedures for experimentally changing the cell parameters, changing the connections between cells, and manipulating the simulation windows. Suggestions of simulation parameters that provide illustrative behaviors for various models are given throughout the remaining sections of this chapter. In the event that the mathematical description of the behavior of a particular model seems unclear, it may be useful to run the simulations before attempting a more detailed study of the mathematical treatment.

8.2 Two-Neuron Oscillators

In this section, we consider models that mimic the behavior of a system of two coupled oscillators in an attempt to understand some of the ways in which biological oscillators may influence one another. Each oscillator can represent a single neuron or a network of cells that collectively function as an oscillator. We first present phase equation models that do not depend on the oscillator structure so that the model results apply to both single cell and network oscillators. We show that these models are simple enough to be analyzed mathematically yet complex enough to capture some of the underlying principles that govern the behavior of a two-oscillator network. Next, we briefly mention a modeling option that uses higher-order systems of equations and incorporates more information about the oscillator structure.

8.2.1 Phase Equation Model of Coupled Oscillators

First consider a general mathematical model due to Rand, Cohen, and Holmes (1988) for a network of two oscillators where each is treated as a simple biological oscillator, ignoring the structure of the oscillation and the mechanisms that produce it. In actuality, the behavior of each oscillatory neuron or network oscillator is determined by a multitude of parameters that can be used to represent the state of the oscillator at any given time. Due to the cyclic behavior of each oscillator, if we draw an orbit in the parameter space that shows how the parameters change with time, the oscillator eventually returns to the same state. This type of orbit is known as a limit cycle. In addition, for small perturbations away from the orbit in the parameter space, the orbit of the oscillator returns to this cycle; that is, the orbit is locally asymptotically stable as shown in Fig. 8.1. These assumptions allow us to test assertions regarding the coordinating system while knowing little about the individual oscillators.

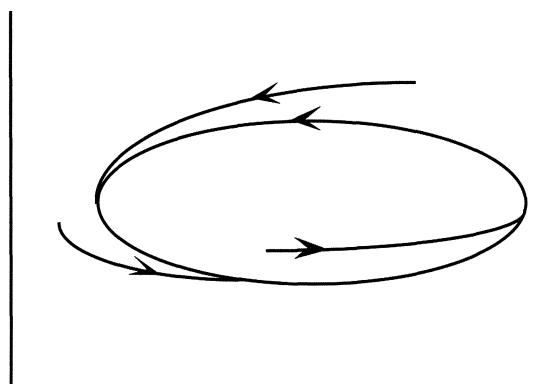


Figure 8.1 Locally asymptotically stable limit cycle in two-dimensional space.

Since we assume that each of the two oscillators in this model can be represented by a structurally stable dynamical system that exhibits a locally asymptotically stable limit cycle, the behavior of each can be represented by a single variable, $\theta_i(t)$ for $i \in \{1, 2\}$, which specifies the position of the oscillator around its limit cycle at time t ; that is, $\theta_i(t)$ specifies the phase of the limit cycle. In addition, we rescale $\theta_i(t)$ so that it flows uniformly around the limit cycle taking values from 0 to 2π radians over one cycle. Thus, $\theta_i(t)$ is proportional to the fraction of the period that has elapsed and the behavior of each oscillator is characterized by the differential equation

$$\dot{\theta}_i(t) = \omega_i, \quad (8.1)$$

where ω_i is the constant frequency of the oscillator, and $2\pi/\omega_i$ is the period. We use modular arithmetic so that $\theta_i(t)$ always lies between 0 and 2π , and the solution to Eq. 8.1 is

$$\theta_i(t) = (\omega_i t + \theta_i(0))(\text{mod } 2\pi), \quad (8.2)$$

where $\theta_i(0)$ is the initial value of θ_i . When two such oscillators are coupled, we obtain a system of equations

$$\dot{\theta}_1(t) = \omega_1 + h_{12}(\theta_1, \theta_2) \quad (8.3)$$

$$\dot{\theta}_2(t) = \omega_2 + h_{21}(\theta_2, \theta_1), \quad (8.4)$$

where $h_{ij}(\theta_i, \theta_j)$ represents the coupling effect of the j th oscillator on the i th oscillator. This coupling term must be 2π periodic since we would like the rate of change to depend only on the oscillator phase and not on the number of cycles that have already occurred or the amplitude of the oscillation. Since the behavior of each cell is described using only the phase, the coupling also depends only upon the phase.

It is often convenient to define a quantity

$$\phi(t) = \theta_1(t) - \theta_2(t) \quad (8.5)$$

which represents the difference between the phases or the phase lag of oscillator 2 relative to oscillator 1. Note that like θ_i , $\phi(t) \in [0, 2\pi]$. Combining Eqs. 8.3 and 8.4, we obtain

$$\dot{\phi}(t) = \dot{\theta}_1(t) - \dot{\theta}_2(t) \quad (8.6)$$

$$= (\omega_1 - \omega_2) + (h_{12}(\theta_1, \theta_2) - h_{21}(\theta_2, \theta_1)). \quad (8.7)$$

Much of the analytical work to date on coupled oscillators deals with interactions that depend only on the difference between the phases. Consider the simplest case where h_{ij} depends only on the phase lag, and in addition $h_{ij}(\theta_i, \theta_j) = 0$ when the phase lag between the i th and j th oscillators is zero. This is known as *diffusive coupling*. For example, if we let $h_{ij} = a_{ij} \sin(\theta_j - \theta_i)$, we obtain

$$\dot{\phi}(t) = (\omega_1 - \omega_2) - (a_{12} + a_{21}) \sin(\phi(t)). \quad (8.8)$$

This equation can be solved exactly, but let us consider those solutions where 1 : 1 phase-locked motion occurs; that is, consider the situation where the phase lag between the two oscillators remains constant. This type of motion occurs when $\dot{\phi}(t) = 0$. Equating the right-hand side of Eq. 8.8 to zero, we obtain

$$\phi = \arcsin\left(\frac{\omega_1 - \omega_2}{a_{12} + a_{21}}\right) \quad (8.9)$$

which has zero, one, or two solutions depending on the value of the ratio of the frequency difference $\omega_1 - \omega_2$ to the net coupling strength $a_{12} + a_{21}$. When the net coupling is small enough, the ratio has absolute value greater than unity. Since the sin function takes values only between -1 and 1 , there is no solution to Eq. 8.9 in this case, and the oscillators will drift with respect to one another. As the coupling is increased, a critical value is reached where the frequency difference is equal to the net coupling strength. At this point, the absolute value of the ratio is 1 , and the system goes into phase-locked motion. If the net coupling is positive or excitatory, then the faster oscillator leads the phase-locked motion by a phase of between 0 and 90 degrees. If the net coupling is negative or inhibitory, the slower oscillator leads by between 90 and 180 degrees.

8.2.2 Simulation Parameters

To interactively study some of the phenomena exhibited by this model, create a simulation of two coupled biological oscillators using the *CPG* tutorial. Begin by changing the strengths of all of the connections to zero so that the simulation environment consists of four isolated cells. We would like the cells labeled “L1” and “R1” to function as two biological oscillators that can be simulated by supplying a constant current injection to the somata of these two cells. Bring up the windows that allow the user to interactively change the inputs to these two cells by clicking on the appropriate toggles under the heading *inputs*. Set the injection current to the soma of L1 to $0.00025 \mu A$ with zero delay and a duration that is greater than the number of steps for the simulation such as 200 msec . The result is a constant current injection throughout the duration of the simulation which causes the continuous firing of the cell. Likewise, create an injection current of $0.0003 \mu A$ to the soma of R1 in the same manner. Make sure that all other possible inputs, including those to L2 and R2, are set to zero current. Don’t forget to hit the “Return” key after any change is made within a dialog box and to click on the *APPLY* button after all of the changes have been made for a single desired input current. After the above changes have been made, click on the *STEP* button to run the simulation. It is important to remember that we are assuming that these two cells are intrinsically oscillatory so that they oscillate continuously, where R1 has a slightly greater intrinsic frequency due to the larger injection current.

Now add coupling between the two cells, starting with small mutually excitatory coupling. This can be done by setting the coupling strength of the connection from L1 onto R1

to 5 and making sure that the toggle for that connection is in the excitatory mode. Set the strength of the connection from R1 onto L1 to 5 in a similar manner. After running the simulation, the graphs reveal that the two oscillating cells are drifting with respect to one another. In other words, the coupling is too weak to cause phase-locked behavior. Now gradually increase the coupling strength between the two cells by increments of 5, clicking on the RESET and STEP buttons after each change in order to observe the behavior of the system. As the coupling strength is increased, eventually the system exhibits behavior that converges to phase-locked oscillations where the cell with the greater intrinsic frequency, namely, R1, leads. This causes the frequency of oscillation of the system to be greater than the intrinsic frequency of L1. Now repeat the process outlined above for inhibitory connections, beginning with weak mutually inhibitory coupling and gradually increasing the coupling strength. In this case, stronger coupling strengths are required for phase-locked behavior due to the nature of the inhibitory connections in the simulation environment as explained in the text of the help selection under MODEL PARAMETERS. During the phase-locked oscillations, the cell with the slower intrinsic oscillation will lead, causing the frequency of the system to be smaller than the intrinsic frequency of R1.

8.2.3 Initial Conditions

In the qualitative analysis of any system of differential equations, it is important to consider the role of the initial conditions since the system behavior depends on these initial values. Consider a CPG simulation architecture that demonstrates the crucial role of the initial conditions for the two-oscillator model presented in Sec. 8.2.1. Begin by changing the current injections to L1 and R1 so that they are identical, say, $0.00015 \mu A$. In this case the two oscillators have identical intrinsic frequencies; that is, $\omega_1 = \omega_2$. Since the frequency difference $\omega_1 - \omega_2$ is zero, Eq. 8.9 indicates that there are two solutions that provide phase-locked behavior, $\phi = 0$ and $\phi = \pi$. Thus, the two oscillators may be phase-locked with no phase lag so that they demonstrate synchronous behavior, or they may be phase-locked 180 degrees out of phase. We can obtain these two types of behavior by varying the initial conditions of the system.

First, set the delays for both cells to zero and make sure that the durations are set to 200 msec. Begin with symmetric mutually excitatory coupling with the coupling strength of each connection set to 20. Run the simulation and observe that the two cells begin firing in phase due to their identical initial values and continue to fire in phase due to the symmetry of the network. Next change the delay of R1 to 15 msec. In this case, we are effectively changing the initial conditions of the system so that we begin with two cells that are firing out of phase at time $t = 15$ msec. As predicted for the case where the intrinsic frequencies of the two oscillators are identical, the cells still exhibit phase-locked motion; however, they fire out of phase. This demonstrates how important initial conditions may be for the production of different temporal patterns of activity in these simplified models. However,

the initial conditions do not play such a crucial role in biological system where noise and inherent differences prevent any two oscillators from having identical frequencies.

8.2.4 Synaptic Coupling

Naturally, the phase equation model described above is not sufficient for predicting or studying all possible types of behavior exhibited by two coupled neurons or network oscillators. In particular, the choice of diffusive coupling may be limiting. Although diffusive coupling probably models the behavior of electrical coupling between multiple cells accurately, in general we would not expect the synaptic interactions between two neurons to depend exclusively on the difference between the phases. For example, with diffusive coupling a phase difference of zero results in a coupling term with a value of zero so that two neurons that are exhibiting identical behavior have no influence on one another. Because this does not seem biologically plausible for synaptic connections between neurons, Ermentrout and Kopell (1990) consider a model that differs slightly from the previous one and contains coupling terms that behave more like synaptic coupling between cells. The assumptions for this coupling hold true for Hodgkin–Huxley-like neural models in which the coupling is due to voltage only (Ermentrout and Kopell 1990). In the following description of the model, we assume that the oscillators are identical for ease of analysis.

The equations for this model are

$$\dot{\theta}_1 = \omega_1 + p(\theta_2)r(\theta_1) \quad (8.10)$$

$$\dot{\theta}_2 = \omega_2 + p(\theta_1)r(\theta_2), \quad (8.11)$$

which are simply Eqs. 8.3 and 8.4 with $h_{ij}(\theta_i, \theta_j) = p(\theta_j)r(\theta_i)$, where p is a periodic smooth pulse function and r plays a role that is analogous to that of a *phase response curve*. A phase response curve, or PRC, for a given oscillator and a given brief stimulus is determined experimentally by stimulating the oscillator and waiting until the system relaxes back to its oscillation with a shift in phase. The PRC can be represented by a function $\hat{r}(\theta)$ that gives the phase shift and depends upon the phase θ at which the stimulus is administered. Consider the situation where the relaxation time is short relative to the time between stimuli τ . Let θ^k be the phase just before the k th stimulus so that we obtain a sequence of phases $\{\theta^k\}$ that are related by the difference equation

$$\theta^{k+1} = \omega\tau + \theta^k + \hat{r}(\theta^k), \quad (8.12)$$

where ω is the natural frequency of the oscillator. We can rewrite Eq. 8.12 as the differential equation

$$\dot{\theta} = \omega + \delta(t \bmod \tau)\hat{r}(\theta), \quad (8.13)$$

where δ is the Dirac delta function that provides a pulse stimulus at intervals a time τ apart. According to Ermentrout, since real stimuli are not instantaneous, it is reasonable to replace

δ with the smooth distributed stimulus function p . In doing so, we obtain an equation analogous to Eqs. 8.10 and 8.11, where r represents the phase response curve \hat{r} . That is, the coupling mimics the effects of coupling two oscillators via their PRC's.

Ermentrout and Kopell show that for this type of coupling, the phase equation model exhibits qualitative behavior that is similar to that of more complex non-phase equation models such as those discussed briefly in the following section. In particular, as the coupling strength increases, the system may go from phase drift to phase-locking and finally to a phenomenon known as *oscillator death*. Oscillator death corresponds to a critical point solution for the system since the oscillator remains at a constant state in the parameter space. In addition, the critical point must be asymptotically stable in order for the oscillator death to be robust. Otherwise, small perturbations would result in a return to oscillatory behavior. This type of behavior cannot occur in the model defined by Eqs. 8.3 and 8.4 since no asymptotically stable critical point exists for that system. However, Ermentrout and Kopell prove that for synaptic coupling, oscillator death is a robust phenomenon for sufficiently large interactions. In addition, they show that the replacement of the stable limit cycle by a stable critical point can also occur in chains of oscillators with coupling between nearest neighbors like those discussed in Sec. 8.3.1. This fact is of interest for the study of models of locomotion that are composed of chains of coupled oscillators such as the Rand, Cohen and Holmes (1988) model for lamprey locomotion mentioned earlier.

8.2.5 Non-Phase Equation Models

The greatest strength of the two models discussed above is their relative simplicity. Phase equation models that allow a single equation to represent the dynamics of an individual cell make it possible to conduct a qualitative analysis of the dynamics of the system. This type of mathematical analysis is much more difficult in higher-order systems of equations. For example, consider the Hodgkin–Huxley model for the biophysics of the squid giant axon, described in Chapter 4. In this model, a system of four differential equations (Eq. 4.1 and Eqs. 4.11–4.13) is used to characterize the ion concentration changes that result in the production of an action potential in a single oscillatory neuron. Thus, a system of eight equations with the appropriate coupling terms is required to capture the behavior of two coupled neurons. Unlike the reparameterized phase equation model, this system of eight equations contains useful information about the manner in which specific parameters change during the oscillatory activity. However, due to the large number of equations required, more detailed models are difficult to study analytically, and simulations must be used to numerically reproduce behavior. Such simulations allow the user to interactively observe the output of the model in an attempt to understand the interactions of the neurons within the system by studying the relationships among the model parameters. The CPG simulation environment for GENESIS that is discussed throughout this chapter is an implementation of the Hodgkin–Huxley equation model. For this reason, the range of behavior demonstrated

by the simulation is more complex than the behavior of the phase equation model of coupled oscillators discussed above; however, some basic interactions can be understood in the context of the phase equation model.

8.3 Four-Neuron Oscillators

Now that we have discussed some of the basic principles that govern the behavior of two coupled oscillators, we would like to apply these ideas to slightly larger networks. As previously mentioned, larger networks are more difficult to analyze mathematically since they require more equations, and thus we will eventually rely on simulations in our attempt to understand their behavior. In this section we examine chains of four oscillators with nearest-neighbor coupling as well as more complex architectures with various types of coupling among four oscillatory cells. We attempt to explore some of the basic ideas used to formulate the vertebrate CPG models mentioned in Sec. 8.1. In particular, the discussion of chains of four coupled oscillators that follows provides insight into various models for lamprey locomotion and the locomotion of some types of fish. These models are inspired by the fact that the waves of muscle contractions that produce locomotion in these organisms are induced by periodic bursts of activity in the ventral roots that emerge at each segment of the spinal cord. The phase lag in activity between segments is proportional to the distance between the points, indicating a constant speed traveling wave of contractions (Rand et al. 1988, Kopell and Ermentrout 1986). Following the discussion of chains of coupled oscillators, we construct networks capable of producing patterns of activity that mimic various gaits for tetrapod locomotion. In these gait simulations, the phase relations among the various oscillators in the CPG model are important since they represent the phase relations among the limbs during locomotion.

8.3.1 Chains of Coupled Oscillators

We begin by extending the concepts outlined in Sec. 8.2.1 in order to develop a phase equation model for a chain of four coupled oscillators. The following treatment appears in Rand et al. (1988) in a more general form for chains of n oscillators. We would like the chain to have only nearest-neighbor coupling with no long-range connections. Thus, Eqs. 8.3 and 8.4 are extended to a system of four differential equations of the form

$$\dot{\theta}_1(t) = \omega_1 + h_{12}(\theta_1, \theta_2) \quad (8.14)$$

$$\dot{\theta}_2(t) = \omega_2 + h_{21}(\theta_2, \theta_1) + h_{23}(\theta_2, \theta_3) \quad (8.15)$$

$$\dot{\theta}_3(t) = \omega_3 + h_{32}(\theta_3, \theta_2) + h_{34}(\theta_3, \theta_4) \quad (8.16)$$

$$\dot{\theta}_4(t) = \omega_4 + h_{43}(\theta_4, \theta_3). \quad (8.17)$$

As before, we assume diffusive coupling and let $h_{ij} = a_{ij} \sin(\theta_j - \theta_i)$. However, we use a uniform coupling strength $a_{ij} = a$ for ease of analysis. Again, we define the quantity

$$\phi_i(t) = \theta_i(t) - \theta_{i+1}(t) \quad (8.18)$$

to represent the phase lag between oscillators for $i \in \{1, 2, 3\}$. So, our system of equations can be represented by the vector equation

$$\dot{\phi}(t) = \Omega + AS(t), \quad (8.19)$$

where

$$\phi(t) = \begin{bmatrix} \phi_1(t) \\ \phi_2(t) \\ \phi_3(t) \end{bmatrix}, \quad (8.20)$$

$$\Omega = \begin{bmatrix} \omega_1 - \omega_2 \\ \omega_2 - \omega_3 \\ \omega_3 - \omega_4 \end{bmatrix}, \quad (8.21)$$

$$A = a \begin{bmatrix} -2 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -2 \end{bmatrix}, \quad (8.22)$$

and

$$S = \begin{bmatrix} \sin(\phi_1(t)) \\ \sin(\phi_2(t)) \\ \sin(\phi_3(t)) \end{bmatrix}. \quad (8.23)$$

As in the previous sections, we are interested in 1 : 1 phase locked motion where the phase lags remain constant; therefore, we would like to find solutions to the equation $\dot{\phi} = 0$. Note that $\dot{\phi} = 0$ when $S = -A^{-1}\Omega$, and we know that

$$A^{-1} = \frac{-1}{4a} \begin{bmatrix} 3 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 3 \end{bmatrix}. \quad (8.24)$$

If we assume that there is a smooth gradient in the intrinsic frequency of oscillation along the cord, then the frequency difference along the chain is constant, and $\omega_1 - \omega_2 = \omega_2 - \omega_3 = \omega_3 - \omega_4 = c$. In this case,

$$\Omega = c \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad (8.25)$$

and substituting, we obtain

$$S = \frac{c}{2a} \begin{bmatrix} 3 \\ 4 \\ 3 \end{bmatrix}. \quad (8.26)$$

Since the entries of the vector S are sin functions,

$$\left| \frac{c}{a} \right| \leq \frac{1}{2} \quad (8.27)$$

is a necessary and sufficient condition for the existence of 1 : 1 phase-locked motion. When this condition holds, the resulting motion is frequency-locked at the average frequency of the uncoupled oscillators (Rand et al. 1988). The oscillations create a traveling wave of activation along the chain unless the oscillators all have identical intrinsic frequencies. When the frequencies are identical, there is no phase difference along the chain, $c = 0$, and the oscillators behave synchronously with the exception of those at each end of the chain. The oscillators at the two ends of the chain are subject to the influence of only one neighboring cell, and thus behave slightly differently than the internal cells of the chain that are influenced by two neighbors. Kopell and Ermentrout extend this model by replacing the diffusive coupling with the synaptic coupling described in Sec. 8.2.4. Their model generates a constant speed traveling wave of contractions without the presence of a gradient of oscillator frequencies along the cord (Kopell and Ermentrout 1986).

8.3.2 Simulation Parameters

To observe the behavior described in the section above which is similar in principle to the behavior of a swimming fish (albeit with few segments), create a simulation of a chain of biological oscillators with nearest-neighbor coupling using the *CPG* tutorial. First, set all of the connection strengths to zero so that the simulation environment consists of four isolated cells. To simulate the behavior of oscillatory cells with identical intrinsic frequencies, set the injection current to the soma of each cell to $0.0002 \mu A$ with zero delay and a duration that is greater than the number of steps for the simulation. Set all other inputs to zero, and be sure to click on **APPLY** after all of the changes have been made for a desired injection current. Click on the **STEP** button to run the simulation, and check to be sure that all four cells are oscillating in an identical manner.

Now build the connections required for a chain of four oscillators with nearest-neighbor coupling as shown in Fig. 8.2. Make sure that the connections are excitatory with identical connection strengths. The choice of connection strength is not important at this point; a connection strength of around 20 will suffice. When you click on **STEP** to run the simulation, you should discover that the oscillators in the chain oscillate in synchrony with the exception of the oscillators on the end of the chain which behave differently due to the end conditions.

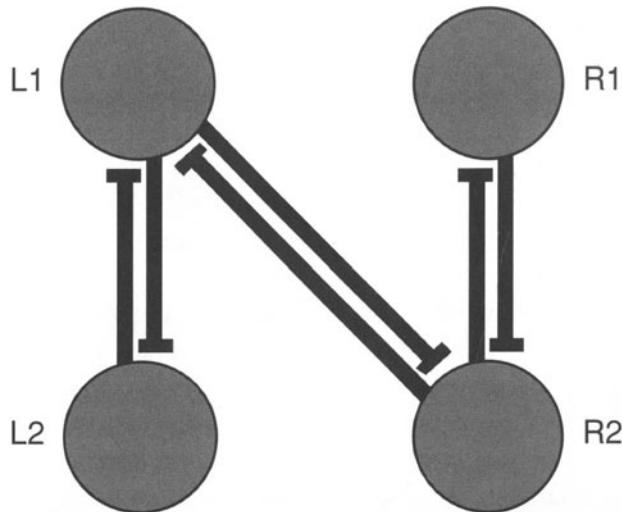


Figure 8.2 Simulation architecture for a chain of four oscillators with nearest-neighbor coupling. All connections are excitatory with identical strengths.

This is the behavior predicted by the model in the case when there is no difference in oscillator frequencies along the chain so that $c = 0$.

Next change the injection currents to the cells causing the cells to have varying intrinsic frequencies with a constant difference in frequency along the chain. For example, you might use values of 0.00035, 0.0003, 0.00025, and 0.0002 μA as the injection inputs along the chain. In this case, the model predicts a traveling wave of activation provided that the value of the connection strength is large enough. Begin with very small values for the identical connection strengths and gradually increase the value until you see a wave of activation. It may be difficult to see the wave since each one begins before the previous wave has reached the end of the chain.

It is also possible to obtain a wave of activation in a chain of neurons simply by causing the first cell in the chain to fire. Each cell activates the next so that the activation moves down the length of the chain. To simulate this type of activity, set the injection input to the soma of the first cell in the chain to 0.0002 μA for a duration of several milliseconds, and set all other possible inputs to zero. This will cause a single wave of activation; however, this model requires outside control for continuous behavior since multiple waves require repeated inputs to the first oscillator at the desired temporal intervals. For this reason, the behavior is less robust than that of the model described above which uses intrinsic oscillators to create continuous waves of behavior in the absence of any external influence to the CPG.

8.3.3 Modeling Gaits

During tetrapod locomotion, interlimb coordination causes different combinations of limbs to be on or off the ground at the same time as the gait of an animal varies. Each limb moves through a step cycle that consists of a swing phase and a stance phase. During the swing phase, the limb is lifted and brought forward, mostly due to the effort of the flexor muscles. During the stance phase, extensor activity is dominant and provides force to thrust the animal forward (Grillner 1981). The most common mode of coordination among limbs is strict alternation between the two limbs of the same girdle; that is, the movements of the two hindlimbs alternate with each other, and the movements of the two forelimbs alternate with each other. These are known as the alternating gaits which include the walk, pace and trot and are seen in a variety of organisms from the millipede to humans. The differences among these three gaits depend on the timing between the forelimbs and hindlimbs. We refer to all other gaits as non-alternating gaits which include the gallop, canter, half bound, and leaping gaits. Most animals prefer to use different gaits at different velocities, but the gait is not directly linked to the speed (Grillner 1981).

We want to model the patterns of limb coordination required for some of the various gaits mentioned above. In these simulations we let one intrinsic oscillator represent the population of cells that controls the cyclic flexor–extensor activity of a single limb. To begin, click on the DEFAULTS button in the main control panel to return to the default architecture provided with the *CPG* simulation. In this default network architecture, the cell labeled L1 represents the left forelimb, R1 represents the right forelimb, L2 represents the left hindlimb, and R2 represents the right hindlimb. Note that all connection strengths are set to a value of 100. Hit STEP and observe the pattern of activation that emerges from the network circuitry. As shown in Fig. 8.3, this pattern of activity is the sequence observed in a walking gait. The lateral symmetric inhibitory connections between the two forelimbs and between the two hindlimbs encourage the alternating phase-locked behavior required for all alternating gaits. The excitatory connections help maintain a robust pattern of activation where each oscillator stimulates the next one in the sequence. The remaining inhibitory connections discourage the oscillators from becoming active out of turn. Also, note that the delays are set to values that initialize the simulation with the sequence of activation required for a walking gait. Experimenting with the delays reveals that a variety of initial conditions will lead to this same pattern of behavior although the amount of time required for convergence to phase-locked oscillations varies for different initial conditions. It is important to make sure that the delays are not all identical since beginning the oscillators in phase results in completely synchronous firing due to the symmetry of the network.

Now, alter the architecture so that the vertical connections between L1 and L2 and between R1 and R2 are inhibitory, and make all of the cross-connections excitatory. In addition, change the delays so that L1 and R2 have delays of zero, and L2 and R1 have delays of 15. Click on RESET and then run the simulation. This pattern of activity mimics

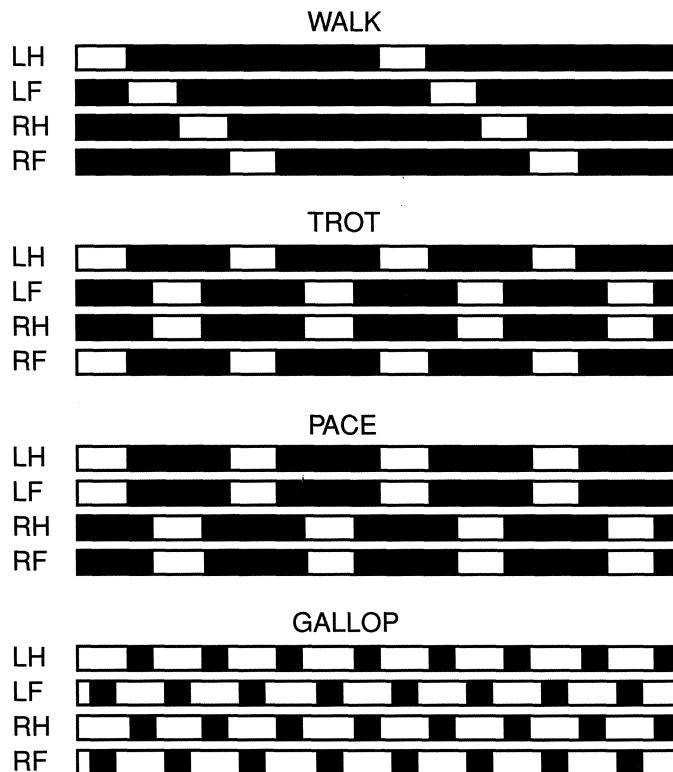


Figure 8.3 Idealized diagram of the stepping movements of the cat for different characteristic gaits. Open bars represent a lifted foot, and closed bars denote a planted foot. Figure adapted from Pearson (1976).

that required for a trotting gait. A similar architecture, where the roles of the left and right forelimbs are switched, provides the pattern of activity required for a pace. In some of the exercises that follow we outline changes in oscillator frequency, initial conditions, and coupling which provide additional gaits and demonstrate some of the ways that these factors can influence the network activity.

In each case, there are many other architectures that will exhibit the same patterns of activity as those mentioned for the various gaits described above and in the exercises. At present, there is little known regarding the mechanisms employed by CPGs in behaving animals. Thus, it is impossible to determine which architectures most closely model these biological mechanisms. Experiments that study CPG-produced locomotion in the cat use treadmill speed to demonstrate that sensory input can lead to gait changes. The stimulation of the mesencephalic locomotor region in these experiments shows that descending drive from higher centers of the nervous system can lead to gait changes as well. Therefore, it seems likely that biological systems may use various methods or combinations of methods

for changing gaits.

8.4 Summary

We are not suggesting that the models and simulations described in this chapter accurately describe how CPGs produce observed patterns of behavior. It is certain that the actual mechanisms are much more complicated and that feedback from proprioceptors and influence from higher centers of the nervous system play an enormous role in the selection and maintenance of robust patterns of activity. However, we believe that this discussion demonstrates how mathematical models and simulations can help us learn to ask the correct types of questions when probing biological systems, and perhaps provide insight into the complex interactions involved in the production of repetitive motor activity.

8.5 Exercises

1. Consider a simulation architecture in which L1 and R1 are identical where each has an input of $0.00015 \mu A$, a delay of zero, and a duration of 200 msec . Implement symmetric mutually excitatory coupling with connection strengths of 1500, and gradually increase the strengths by increments of 100. How might one explain the observed behavior? What happens when the symmetry is broken; that is, what happens when the coupling strengths are unequal, the injection currents are unequal, or the initial conditions differ?
2. It is interesting to note that a fish is capable of swimming backwards when placed in a corner (Grillner 1974). Consider the architecture and parameters for a chain of oscillators described in Sec. 8.3.2. Try experimenting with the parameters to cause the wave to travel in the opposite direction and consider the implications for possible mechanisms for reversing the direction of the wave propagation in a fish.
3. Figure 8.3 shows that the pattern of activity for the idealized gallop is identical to that for the walking gait except that the speed is more rapid. Alter the default architecture provided for the walking gait to create a faster gait that is comparable to a gallop. Begin by simulating an increase in the frequency of the intrinsic oscillations via an increase in the current injection.
4. In a true gallop, there is a lag between the planting of the left forelimb and the right forelimb. Alter the architecture for the idealized gallop that was developed in the preceding question to obtain this asymmetry. Begin by introducing asymmetries in the connection strengths.

5. In Sec. 8.3.3 we mention that the differences between the various alternating gaits depend on the timing between the forelimbs and hindlimbs. In fact, the transition from walking to trotting is continuous. As the locomotion speed increases, each forelimb begins to step before the opposite hindlimb touches the ground until the opposite legs step at the same time resulting in a trotting gait (Pearson 1976). Try to mimic the transition from a walk to a trot by making incremental alterations to the architecture and initial conditions.
6. In various experiments, both picrotoxin and strychnine, which block inhibitory synapses, induce changes in the motor output pattern of several different organisms (Rand et al. 1988). These changes involve a transformation from an out-of-phase mode of oscillation to an in-phase mode. Delete the inhibitory synapses in the various gait simulation architectures discussed in this chapter and in the exercises above. In each case, do the effects of the blocked inhibition support the choice of architecture or suggest that an alternate choice might be more biologically plausible?
7. Use the insight gained from the various models in this chapter and the gait simulation architectures discussed above to create architectures that simulate the non-alternating gaits such as the canter, leap, and half bound.

Chapter 9

Dynamics of Cerebral Cortical Networks

ALEXANDER PROTOPAPAS and JAMES M. BOWER

9.1 Introduction

Previous chapters in this volume have considered detailed models of single cells and small networks of cells. In this chapter we consider a large scale multicellular model of the mammalian olfactory cortex. The simulation consists of three distinct neuronal populations of 135 cells each for a total of 405 interconnected neurons. With this simulation, we will explore the possible physiological basis for experimentally recorded electroencephalographic patterns in this cortex.

When constructing a model of any neural system, one must always strike a balance between biological realism and computational efficiency. This is especially true in the case of network models. The real piriform cortex of a rat, for example, contains on the order of 10^6 neurons (Haberly 1990). Even when the complexity of individual neurons is reduced, it is still not possible to simulate all the neurons found in this network. Accordingly, the modeler is always faced with determining the level of detail necessary to explore and illuminate the particular physiological and computational properties of these networks. An important question becomes: how do the physiological properties or computational capabilities of a cortical network scale with the number of neurons? In this tutorial we demonstrate how at least a rough understanding of network behavior can be obtained with a quasirealistic model of cerebral cortex. First, however, we briefly introduce the piriform cortex.

9.2 Piriform Cortex

The piriform cortex is the primary olfactory cortical area in the mammalian brain. For the last several years we have been constructing realistic models of this network (Wilson and Bower 1992) with the ultimate objective of understanding its role in olfactory object recognition (Hasselmo and Bower 1993, Bower 1995). One motivation for this work is our assumption that this cortex computationally represents a kind of associative memory (Haberly 1985, Haberly and Bower 1989).

Piriform cortex receives its afferent input from the olfactory bulb which itself receives input directly from the nasal epithelium where the olfactory receptors are located. Thus, piriform cortex is quite close to the sensory periphery, and unlike other sensory cortical areas, it does not receive its afferent input through the thalamus. Piriform cortex sends its primary projections to the entorhinal cortex but also connects to the thalamus, olfactory tubercle, superior colliculus, peri-amygdaloid, and has strong reciprocal connections with the olfactory bulb (Haberly 1990). Figure 9.1 illustrates the connections between piriform cortex and related areas.

Piriform cortex is believed to be phylogenetically older than other sensory cortical areas and is commonly referred to as *paleocortex*. It also has a particularly well-defined and somewhat simpler anatomical organization than neocortical regions (Haberly 1985). For example, it has only three layers instead of the six normally found in neocortex. In this sense, it is similar to the hippocampus which also has a trilaminar structure. There are considerable physiological data available on its neurons, their interactions, and on network level responses (Haberly 1990). The detailed internal structure of the piriform cortex is discussed in the context of the model's implementation.

9.3 Structure of the Model

The simulation discussed in this chapter was originally constructed by Matt Wilson when he was a graduate student at the California Institute of Technology. In fact, this simulation served as the initial basis for the construction of GENESIS itself. Portions of the model description below were taken from a paper originally written by Wilson and Bower (Wilson and Bower 1989). The graphical interface was later added to make the simulation user-friendly. Although this model has been used to explore a wide range of cortical behavior (Wilson and Bower 1989, Wilson 1990), including associative memory function (Wilson and Bower 1988, Hasselmo, Wilson, Anderson and Bower 1990), the tutorial version has been simplified for the sake of computational speed and pedagogical ease. In its current manifestation, the model allows you to reproduce experimental EEG patterns and to explore their possible physiological basis. Using later chapters of this book, the user can expand this simulation as he or she wishes.

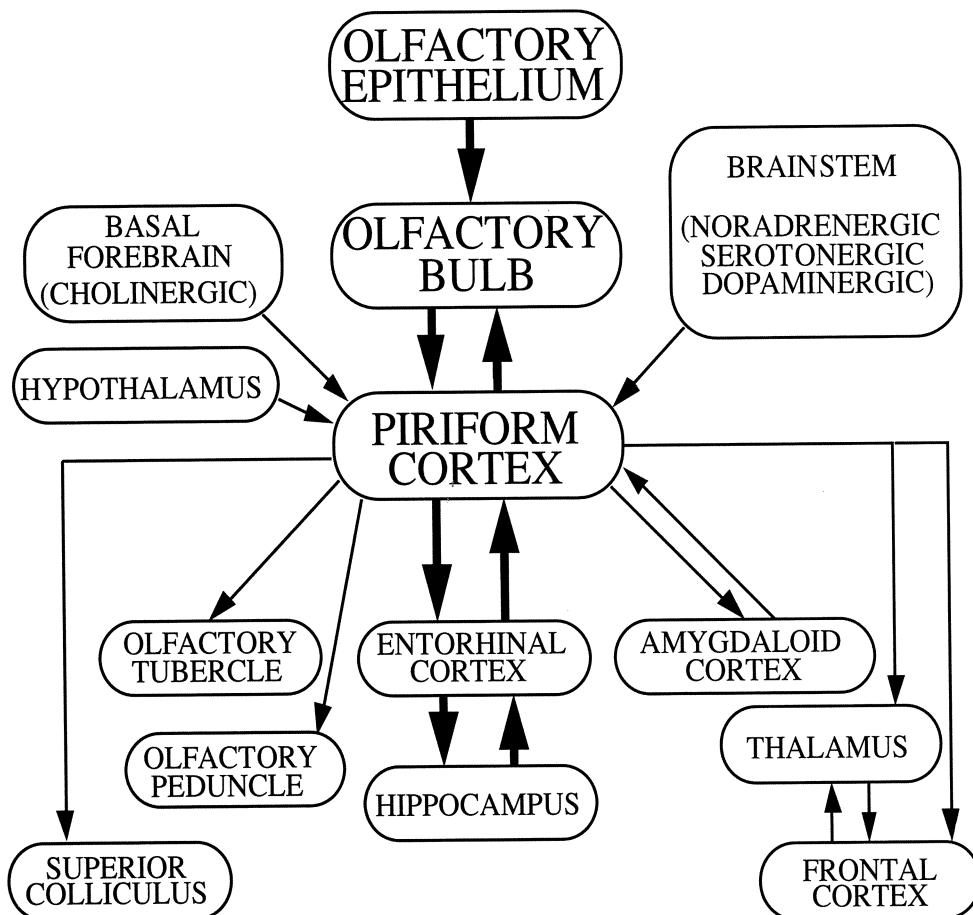


Figure 9.1 Piriform cortex and connected areas. Major pathways are indicated by thick arrows. In the model, only the olfactory bulb to piriform cortex connection is included. The feedback pathway is not incorporated.

9.3.1 Cellular Complexity

Pyramidal cells are the principal cell type in piriform cortex, and are believed to be exclusively excitatory (Haberly and Price 1978, Haberly and Bower 1984). Superficial and deep pyramidal cells form two distinct populations that differ significantly in their physiology (Tseng and Haberly 1989). There are also several populations of nonpyramidal cells or interneurons that can be distinguished on anatomical grounds (Haberly 1983). These neurons appear to be GABAergic and seem to mediate both feedback and feedforward inhibitory effects.

As mentioned previously, computational limitations require that realistic network models be constructed of fewer neurons than are actually found in the brain. The current model is

based on a single population of pyramidal cells plus two populations of inhibitory interneurons. A total of 135 neurons of each type are simulated, yet these neurons are intended to represent the full extent of the actual cortex (approximately $10\text{ mm} \times 6\text{ mm}$). Accordingly, although we simulate neurons individually, the output of each neuron is taken to represent the average activity of a larger group of cells that would normally be in its region. This adjustment for scale is made in the strength of synaptic connections between the cells, and in the number of cells contacted by a particular neuron within the simulated network.

Network models also usually include much simpler representations of neurons than are found in realistic single cell simulations. In the current case, pyramidal cells are modeled using five electrical compartments, whereas interneurons are modeled using only one. We have chosen to model pyramidal cells with five compartments for several reasons. First, in order to accurately model field potentials, it is necessary to distribute synaptic inputs spatially along the dendritic processes of these cells. Secondly, the spatio-temporal distribution of synaptic input along the dendritic tree of the cell may be computationally relevant. Much more realistic multi-compartment models are currently being used to explore this possibility (Protopapas and Bower 1994). Little is lost in modeling inhibitory neurons as single compartments because little is known about the organization of their synaptic input and their small size and radially spanning dendrites preclude them from making significant contributions to the EEG.

In addition to the reduction in the number of compartments for each cell, the membrane properties of these neurons have also been simplified. For example, although experimental evidence indicates that there are a number of Ca^{2+} and K^+ currents in piriform pyramidal cells (Constanti and Sim 1987, Constanti, Galvan, Franz and Sim 1985, Constanti and Galvan 1983) in addition to standard Hodgkin–Huxley Na^+ and K^+ currents, none of these are modeled in any of the cells. Rather, a simple threshold criterion is applied to the membrane potential to generate discrete spike events. The occurrence of spikes is indicated with a spike waveform “pasted” onto the actual membrane potential at the appropriate time. In this way, the computationally expensive details of spike generation are avoided but we are still able to generate the appropriate currents and membrane potentials associated with real action potentials. Similarly, synaptic conductances are modeled neglecting computationally expensive details such as the kinetics of ligand binding, neurotransmitter uptake, etc. Instead, changes in synaptic conductance are modeled as the difference of exponential functions that approximates the shape of EPSPs seen in experimental studies.

9.3.2 Network Circuitry

Primary afferent input enters piriform cortex via the lateral olfactory tract (LOT) projection from the mitral and tufted cells of the olfactory bulb. This is shown in Fig. 9.2. Experimental evidence suggests that this projection is exclusively excitatory (Biedenbach and

Stevens 1969a, Biedenbach and Stevens 1969b, Haberly 1973b, Haberly and Bower 1984) and extremely diffuse or non-topographic. These afferents make excitatory connections to pyramidal cells and feedforward inhibitory cells in layer Ia (Haberly 1985). LOT input is modeled as a set of independent fibers that make sparse connections with pyramidal cells and both types of inhibitory interneurons. In this tutorial, activity along the afferent pathway is random over time. In both the actual cortex and the model, conduction velocities along axons are finite and vary with the axonal type (Haberly 1978). Signals travel along the LOT rostrally to caudally, and are distributed across the cortex via many small collaterals (Devor 1976). In the model, as in the brain, signals proceed along the LOT towards the cortex at a speed of 7.0 m/s . Collaterals leave the main fiber tract at a 45° angle and travel across the cortex at a speed of 1.6 m/s (Haberly 1973a). In the biological cortex there is a diminution of afferent input to pyramidal cells moving rostrally to caudally that is reflected anatomically in the number of synaptic terminals (Price 1973, Schwob and Price 1978), and physiologically in the amplitude of shock-evoked potentials mediated by the afferent system (Haberly 1973a). To simulate this effect in the model, the strength of synaptic input due to afferent signals is exponentially attenuated with increased distance from the rostral site of stimulation.

In addition to excitatory input from the bulb, pyramidal cells within the piriform cortex make excitatory connections with other pyramidal cells across the entire cortex (Biedenbach and Stevens 1969a, Haberly and Bower 1984). The fibers appear to spread out radially from the originating cell and travel rostrally at a speed of 1.0 m/s , and caudally at a speed of 0.5 m/s (Haberly 1973a, Haberly 1978) making local connections on basal dendrites of other pyramidal cells and distant connections on apical dendrites. In the model, fibers originating from pyramidal cells follow the same pattern of interconnectivity and signals are propa-

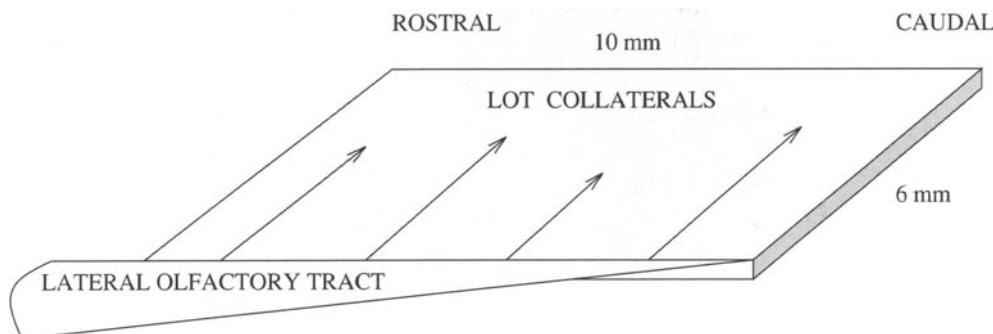


Figure 9.2 The diagram indicates the distribution of afferent input to real and simulated piriform cortex. Input from the olfactory bulb arrives via the lateral olfactory tract which then sends perpendicular collaterals into the cortex that make sparse connections with piriform cells. The number of connections between the LOT and the cortex decreases as one travels rostrally to caudally.

gated along each fiber with the corresponding delays. Simulation scaling considerations as described earlier require that association fiber interconnectivity be greatly increased as compared with that of the actual cortex. As with afferent input, intrinsic excitatory connections are attenuated exponentially with distance from the originating cell.

Experimental evidence suggests the existence of two types of inhibition in the piriform cortex, both of which are incorporated into the model. A well-documented Cl^- mediated feedback inhibition is thought to be generated by local interneurons that receive input primarily from local pyramidal cells as well as some afferent fibers (Biedenbach and Stevens 1969a, Biedenbach and Stevens 1969b, Haberly 1973b, Satou, Mori, Tazawa and Takagi 1982, Haberly and Bower 1984). The outputs of these inhibitory interneurons feed back into nearby pyramidal cells where they activate a significant Cl^- conductance increase at the cell body. A K^+ mediated inhibition is also present in the biological cortex and appears to be generated by local inhibitory interneurons receiving primarily direct afferent input from the LOT as well as some association pathway input from pyramidal cells (Satou et al. 1982, Tseng and Haberly 1986). The outputs of these interneurons generate a long-latency, long-duration hyperpolarizing inhibitory potential in nearby pyramidal cells in the most distal part of the pyramidal cell apical dendrite. In the model, the K^+ mediated inhibition is activated on the apical pyramidal cell dendrite by inhibitory neurons with both feedforward and feedback input. The network circuitry for the model is illustrated in Fig. 9.3.

9.4 Electroencephalography

Because the electroencephalogram (EEG) is the physiological measure that this tutorial attempts to simulate, it merits some explanation. Unfortunately, the EEG is something which has a long history in neurophysiology but whose origins are still debated. In general, physiological measurement techniques can be grouped into those that record the responses of individual neurons, and those that measure the more complex aggregate electrical activity of networks of cells. Single cell electrical recordings can be either intracellular or extracellular and reflect the actual output of single neurons. The origins and significance of aggregate recordings such as the EEG are more difficult to determine. The EEG is identical to extracellular single unit recording in that it measures the field potentials generated in the space around neurons. It differs because EEG recordings represent electrical activity over a wider area of the brain. Typically, EEGs are recorded from an array of electrodes placed on the surface of the brain or even the scalp. In this sense, one may think of the EEG as the average electrical activity of many neurons over a sizable area. The EEG is calculated in the model using an array of 40 evenly spaced electrodes on the surface of the simulated cortex. Recordings from the array are averaged to produce the EEG.

Understanding in detail how extracellular field potentials such as the EEG are related to the activity of collections of single cells is not a straightforward matter. A neuron can be

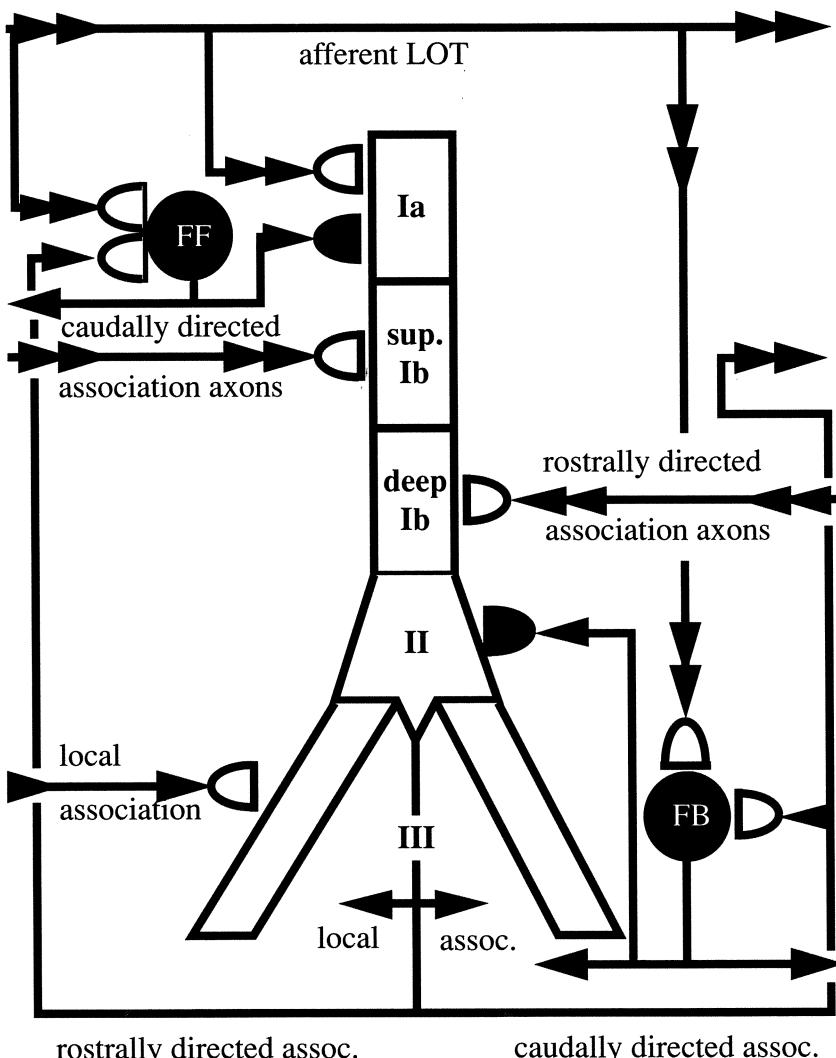


Figure 9.3 Diagram of local model circuitry. The large cell in the center represents a pyramidal neuron. Inhibitory cells are represented by black filled circles. The two types of inhibitory interneurons are distinguished as FF (feedforward) and FB (feedback). Single arrowheads signify local pathways. Double arrowheads are used to show distant pathways. Excitatory synapses are shown as lightly colored cones and inhibitory synapses are black. In the model, the basal dendrites are modeled as a single compartment.

thought of as a very complicated circuit consisting of resistors, capacitors, and batteries (see earlier chapters). Like any circuit, it must obey Kirchoff's current law, which states that the sum of the total current entering and leaving a circuit node must equal zero. In the case of a neuron, this means that if synaptic current, for example, enters the cell at one point, it must leak from another, thereby generating an extracellular current (see Fig. 9.4). An area of the neuron where current is entering the cell is called a current sink. An area where current flows outward is called a current source. Here, we use the “physiologists’ convention” (Sec. 4.3.2) which holds that inward current is negative and outward current is positive. When treating current sources and sinks discretely as we do in the model, the field potential is dependent on these extracellular currents according to the equation:

$$\Phi(\vec{r}, t) = \frac{1}{4\pi\sigma} \sum_{i=1}^n \frac{I_i(t)}{R_i}, \quad (9.1)$$

where Φ is the field potential in volts, $I_i(t)$ is the total current (amperes) from the i th current source into the brain tissue of conductivity σ ($\Omega^{-1}m^{-1}$), and R_i (meters) is the distance of the i th current source from the field point \vec{r} (Nunez 1981). Although this equation is used to calculate field potentials in the model, it is based on the assumption that the brain is a homogeneous conductor, which is only an approximation to biological reality. The important thing to note here is that the size of the field potential increases in amplitude with the magnitude of extracellular current and decreases with distance between current source (or sink) and electrode. This has certain important implications. For example, it suggests that the action potentials of individual neurons often make little contribution to the EEG. Because the extracellular currents produced during spike generation are generally small, the greater magnitude of synaptic currents makes more significant contributions. Since the EEG represents the averaged electrical activity of many neurons, the more synchronous the activity, the stronger the signal will be.

Although understanding the basis for EEG activity is by no means straightforward, many attempts have been made to correlate EEG patterns with certain types of animal and human behavior. EEGs are primarily distinguished on the basis of their frequency. The two frequencies most commonly discussed in the context of the olfactory system are the theta (4–7 Hz) and gamma (30–80 Hz). These types of EEG activity are common, not only in the olfactory system, but throughout the brain. They are also found across species (Ketchum and Haberly 1991). In the rat, hippocampal EEGs in the theta range are prominent during exploratory behavior (Ranck 1973). More recent experiments have indicated that theta-patterned stimulation in the hippocampus is optimal for the induction of hippocampal long term potentiation (Larson, Wong and Lynch 1986, Staubli and Lynch 1987). Theta stimulation has been used to successfully induce LTP in the piriform cortex as well (Kanter and Haberly 1990). Interestingly, the theta rhythm approximates the rate of exploratory sniffing in the rat (Macrides, Eichenbaum and Forbes 1982).

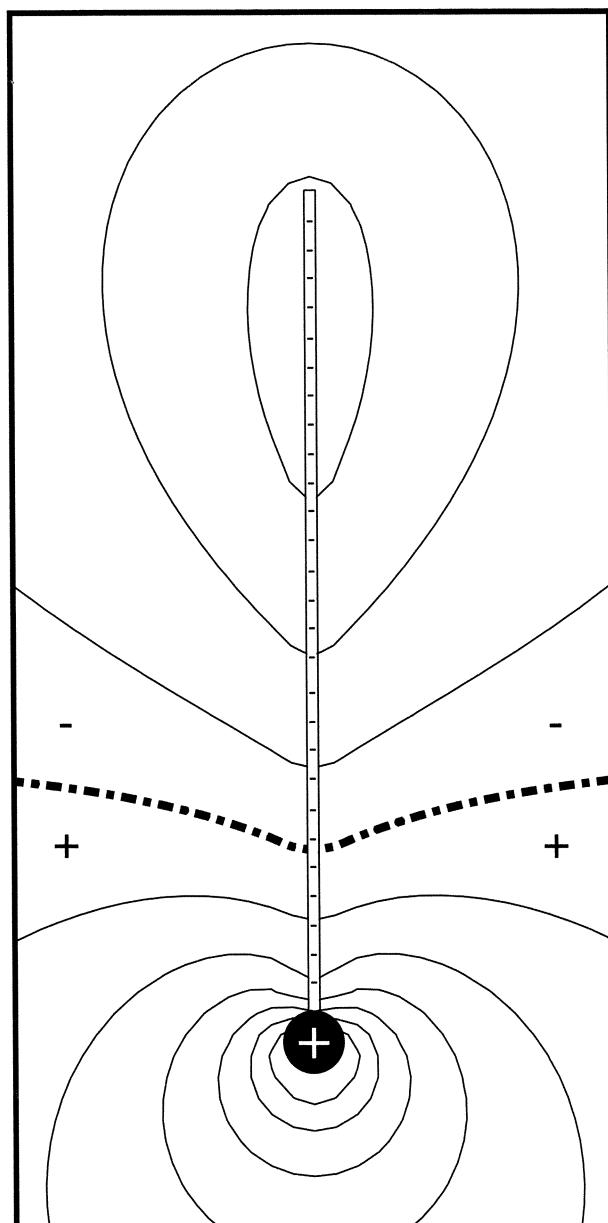


Figure 9.4 A model cell shows the distribution of current sources and sinks and extracellular isopotential contours at the moment when the cell is receiving excitatory input all along its apical dendrite. The thick contour shows the area of zero potential. Contour lines above the zero line give negative potentials. Below the zero line are positive potentials. Minus signs represent current sinks (current flowing into the cell) and the plus sign depicts a current source (current flowing out of the cell).

The gamma frequency (in the 40 Hz frequency range) has recently been the focus of intense experimental and modeling efforts. One of the earliest observations of the gamma frequency was in the olfactory system of the hedgehog in response to an odor stimulus (Adrian 1942). Since then, these oscillations have been found in a variety of cortical areas. Numerous theories have evolved to explain the significance of the gamma rhythm. Researchers have proposed that it is a solution to the binding problem (Gray, Konig, Engel and Singer 1989), a cortical information carrier (Bressler 1990), and even a hallmark of consciousness (Crick and Koch 1990). Although such statements are highly speculative, the ubiquity of these events suggests that the activity underlying the 40–60 Hz oscillations is related in some way to neural computation. The research model on which this tutorial is based was used to explore the possible physiological underpinnings of these oscillations (Wilson and Bower 1991) and led to the conclusion that cortical oscillations do not represent an information code, as suggested in some of the speculations above, but rather reflect the coordination of interneuronal communication within these networks (Bower 1995) A comparison of the EEGs generated by the model and real data is shown in Fig. 9.5.

9.5 Using the Tutorial

As should be clear from the introduction to this chapter, the piriform cortex model that is at the base of this tutorial is quite complex. In fact, this is the most complex tutorial in this book. We have designed an interface for the tutorial that is relatively intuitive to use, however, as you will see, one can quickly become inundated with graphical displays and flashing colors. Any effort to understand all the details of this simulation is certain to be a substantial undertaking. Accordingly, the rest of this chapter should be regarded only as an introduction to some of the features of the tutorial. We encourage you to make further explorations on your own.

9.5.1 Getting Started

If you type “`genesis Piriform`” after changing to the *Scripts/piriform* directory, you will execute the script for the *Piriform* tutorial. Because of the complexity of this model, it will take some time to initialize the simulation, especially on a slow machine. You will notice a multitude of messages appearing in the terminal window from which you started the simulation. You can ignore these for now. They simply provide information about which scripts have been executed. These messages can act as a powerful debugging tool if you have any intention of later modifying the scripts. Once these messages have ended, a large colorful window should appear in the upper left corner of your screen. This is the menu window that also serves as a circuit diagram for the modeled network. In order to decipher the various colors and shapes you see in the window, click on the `help` button at the bottom

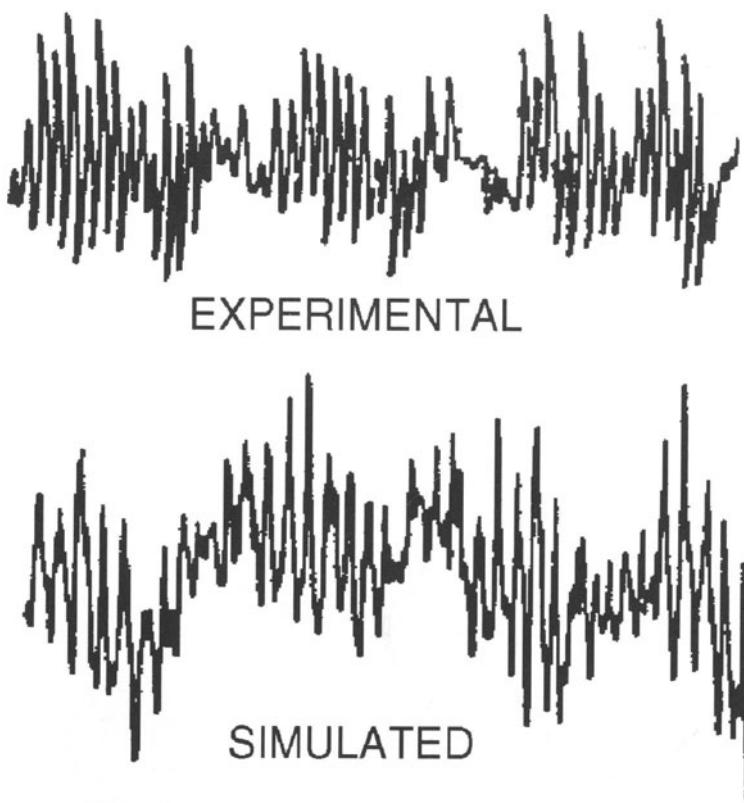


Figure 9.5 Comparison of an EEG measured from rat piriform cortex and data generated using this simulation (Wilson 1990).

of the window. A large text-filled window should appear on the right half of your screen. This window contains an on-line help file which you can scroll up and down using the scroll bar on the left side of the window. Click on the button at the bottom of the help window labeled **legend**. This should generate a window that explains the meaning of the colors and shapes you see in the menu. To hide any of these windows, simply click on the buttons labeled **cancel**.

9.5.2 Generating Simulated Data

In order to generate data using this tutorial, you must first go to a directory to which you have write access. To do this, type “`cd <directory>`” from the window that is running GENESIS, where “`<directory>`” is some directory to which you are allowed to write. Then copy the file called `test_dir` from the tutorial directory to the directory to which you

wish to write your data. The *test_dir* file contains a UNIX shell script that tests to see if the data directory you are trying to create already exists.

The tutorial operates in two modes: PLAYBACK and SIMULATE. In order to generate data, you must be in SIMULATE mode. On the lower left corner of the menu window is the mode toggle labeled either SIMULATE or PLAYBACK. If it reads PLAYBACK, click on it so that it reads SIMULATE. Now click on the **simulate** button next to the toggle. A window also labeled **simulate** should appear over the button. The dialog labeled **step size** allows the user to set the time step of the simulation. As discussed earlier in this text (Chapter 2), there is always a tradeoff in setting step size. A large step size will increase the speed of the simulation but decrease the accuracy of the results, whereas a small step will improve accuracy at the cost of speed. The optimal step size will depend on the speed of your machine, your desire for accuracy, and most importantly, your patience. For now, let's leave **step size** at 0.1 ms . This means that the simulation will generate ten data points for every one millisecond of simulated time. A millisecond of simulated time corresponds to a millisecond of time in the biological network.

The dialog labeled **simulation duration** indicates the length of the simulation run and **data directory name** is the name of the directory to which you will store your simulation data. For now, let the dialog read **default**. The dialog **output rate** contains the sampling frequency for stored data. If **step size** is 0.1 ms and **output rate** is 1.0 ms , this means that for every ten simulated steps, only one value will be stored. This parameter is just as important as **step size** because once you are finished generating data, you will only be able to look at your data with the temporal resolution indicated in this dialog.

Now it is time to run the simulation. In the **simulate** window, click on the button labeled **reset**. This must be done each time you run a new simulation. Now click on **start simulation**. Wait about ten seconds and then click on **simulation status**. A window should appear over the menu that will tell you what percent of the simulation run is complete. This will give you a good idea of how long it will take to generate your data. The simulation status is not automatically updated. It will only update whenever you click on the **simulation status** button in the **simulate** window or the **update** button in the **sim_stat** window. By clicking on the **stop** button, you can freeze the simulation run. By clicking on **continue simulation**, data generation will resume from the point at which it was halted. Now wait until the simulation has finished. The run is complete when the **start simulation** or **continue simulation** button is no longer highlighted and when the **simulation status** window reads 100%.

9.5.3 Initial Look at Simulated Activity

As you will see in later sections, the graphical interface for this tutorial allows you to observe many different aspects of the model's behavior. Before introducing you to the intricacies of the interface, it will be useful to give you a sense for the overall network behavior. To do

this, you should find the three buttons to the right of the pyramidal cell in the menu window labeled **field potential**, **spike count**, and **full cell view**. Each of the graphs associated with these buttons allows you to look at a different measure of global network activity. Click on the button titled **field potential**. Beneath the title bar labeled **fp's generated by synaptic currents** are toggles corresponding to the six termini of the six different synaptic pathways going to the pyramidal cells, and below that a title bar reading **eeg display**. For the moment, just click on the button labeled **eeg**. When you play back the simulation, this graph will show you the simulated EEG.

A second form of network behavior you can monitor is the total spike count over time for any neuronal class in the model. Click on the **cancel** button at the bottom of the **field potentials** window and then click on **spike count**. A menu window should appear over the button listing three options: **pyramidal**, **feedback inhibitory**, and **feedforward inhibitory**. Click on **pyramidal**. A graph titled **spike_activity_pyr_default** should appear next to the menu window. Spike count graphs show the total number of action potentials occurring during a playback step size for all the members of a particular cell class. In this case, we are looking at spike count for the pyramidal cells, but we also could have chosen to count spikes for feedforward and feedback inhibitory cells.

In addition to overall patterns of network activity, the interface allows you to examine the response properties of many individual network components. The details of this capacity are described in later sections of this chapter. For now, look at the pyramidal cell in the menu window and notice that at the center of every cell compartment there is a button indicating laminar position. These buttons allow you to visualize data associated with each compartment. Click on the button labeled **supIb**. The dialog box that appears has two options **network view** and **total synaptic conductance**. Click on **network view**. The new window includes several display options. Click on **synaptic conductance (Na)**. A window containing a two-dimensional array of rectangles should appear in the upper right corner of the screen. It should be labeled **net_pyr_gCA.default**. This window shows the synaptic Na^+ conductance for each superficial Ib layer compartment of each pyramidal cell.

We are finally ready to play back the data we have generated. In order to look at the simulation results, you must put the tutorial in **PLAYBACK** mode. Go to the toggle in the lower left part of the menu window and click on it if it reads **SIMULATE**. It should now read **PLAYBACK**, indicating that you are in **PLAYBACK** mode. Go to the **playback** button at the bottom of the menu window and click on it. The playback window that appears contains many of the same options you saw in the **simulate** window and should therefore be self-explanatory. The only thing to note is the clock dialog at the top of the window which indicates the number of milliseconds of simulated activity that have been played back. Click on the **reset** button and then on the **step** button. The **step** button advances the playback by one time step as specified in the **playback step size** dialog. In order to play everything from start to finish, click on **run**. Do not hide any of the display windows because you will need them for the next section.

9.5.4 Observing Network Behavior

If you followed all the instructions up to this point, you should be observing three different types of simulated results simultaneously. You will first notice the complex oscillatory pattern of the EEG (note the roughly 40 Hz oscillations), and the oscillatory spiking activity of the network. If you now pay close attention to the network window displaying synaptic conductance (labeled `net_pyr_gCA.default`), you will notice waves of synaptic activity propagating from the rostral to the caudal end of the network. This is what we would expect given that the superficial Ib compartment of the pyramidal cell receives synaptic input from rostral pyramidal cells.

In the real brain, the olfactory cortex is known to receive oscillatory inputs from the olfactory bulb (Bressler 1987). Furthermore, these bulbar oscillations occur in the same theta and gamma ranges seen in the olfactory cortex EEG. In this tutorial, input coming from the olfactory bulb is modeled as continuous and random, yet the piriform model still generates the oscillatory patterns found in the piriform cortex when recording from an awake behaving animal. This suggests that the intrinsic properties of the piriform cortex, rather than patterned input, are responsible for these oscillations in the real cortex (Wilson and Bower 1992).

9.5.5 Varying Network Parameters

Over the last several years we have published several papers on what we consider to be the physiological and computational implications of these simulations of the oscillatory structure of the piriform cortex (Wilson and Bower 1992, Bower 1995). We have also modified this simulation to make the model more like neocortex (Wilson and Bower 1991) and used it to replicate the recent oscillation results in visual cortex (Gray et al. 1989). Readers interested in our interpretations are encouraged to read these papers. Here, we briefly introduce you to the way in which you can modify model parameters to further explore the physiological basis of these oscillations.

When one wishes to study the behavior of a model, it is important to be able to vary its parameters. As you have learned, GENESIS was designed to make that relatively easy. Of course, one must still carefully choose which parameters to vary because a biologically realistic simulation as complex as this one is dependent on a multitude of parameters that would take many years to fully explore at random. Typically, in complex simulations of this sort, which parameters are varied is determined either by the level of experimental certainty of the parameter value, or by parameters that are known to vary in the biological system. For example, if a parameter is well defined experimentally, and well known to be invariant, it is less necessary to run a full parameter search (Bhalla and Bower 1993). In the current simulation, for example, the values for conduction velocities in piriform cortex were previously experimentally determined, and most likely remain constant in the context of

normal functioning; therefore, it may be less important to vary conduction velocities except to illuminate their role in normal activity. On the other hand, it is well known that synaptic efficacy varies greatly with the presence of certain neuromodulators. Experiments in piriform cortex have shown that synaptic transmission in the intrinsic excitatory pathways (layer Ib) is significantly diminished following the application of acetylcholine or norepinephrine (Hasselmo and Bower 1992, Vanier and Bower 1993). In addition, carbachol (a drug which mimics the effects of acetylcholine) has been shown to suppress gamma oscillations and intensify the theta rhythm in the piriform cortex of cat (Biedenbach 1966). In the following exercise, we will decrease the synaptic efficacy of excitatory inputs going to layer Ib and study the effects of this change on the EEG.

In this tutorial, the user is allowed to change the **weights** (or efficacy) of synaptic pathways in the model as well as the duration of **channel open times** (this can also be thought of as duration of synaptic current) in the pyramidal cell. To see how this is done, click on one of the synapses on the pyramidal cell. A window should appear over the synapse containing dialogs specifying **weight** and **channel open time**. Synapses on inhibitory cells can have their **weight**, but not their **channel open time** altered.

To examine the effects of changing the synaptic efficacy of the excitatory association pathways in layer Ib, first make sure the tutorial is in **SIMULATE** mode. You will notice that the previous **PLAYBACK** windows will disappear once the simulation is in **SIMULATE** mode. Don't be alarmed by this. It's supposed to happen. Now go to the excitatory synapse on layer supIb and click on it. A window titled **rostral pyr to caudal pyr synapse** should appear over the synapse. The weight value is 1.0 in the dialog labeled from **rostral pyr**. This indicates that the strength of this pathway is 1.0 times the default value. Default values for synaptic strengths are determined by finding synaptic strengths that permit the simulation to replicate physiological activity. Change the weight dialog to 0.05 in order to make the synaptic efficacy 20 times weaker than the default value. Click on the synapse going to layer deepIb. When the window titled **caudal pyr to rostral pyr synapse** appears, go to the weight dialog and enter 0.05. Now click on the button labeled **apply**. Every window associated with a synapse has an **apply** button that incorporates the changes the user has made. This button has to be clicked in only a single window after all the appropriate changes have been made. You can also click on it every time you make changes in a synapse window, but clicking it only once will save you time.

Open the **simulate** window by clicking on the **simulate** button on the control bar at the bottom of the menu window. Go to the **data directory name** dialog and enter **ach** (for acetylcholine). Remember that in the previous simulation run, the name of the data directory was **default**. Click on **reset** and then **start simulation**.

When the simulation is complete, set the mode toggle to **PLAYBACK**. In the playback window, make sure that the **data directory** is set to **default**. Click on the **field potential** button, then click on **eeg**. Go to the **spike count** button and click on it. Then select **pyramidal**. Go back to the **supIb** button and click on it. In the windows that pop

up, choose network view and then synaptic conductance (Na). These are the same displays you were looking at previously. Now click on the playback button at the bottom of the menu window and type ach in the data directory dialog. Go back to the field potentials window and click on eeg again. Then make the same spike count and supIb selections that you made previously for the default data. After you have done this, you will notice that you now have a duplicate set of windows. Note that the last word in each window title indicates the data directory name. This is the case with all display windows used in PLAYBACK mode. The tutorial allows you to display any number of windows from any number of directories simultaneously. Go back to the playback window and click on reset and then run. All the windows should start running simultaneously. Compare the way in which the simulation evolves in the default and acetylcholine case. Is the difference subtle or dramatic?

Now let's look at the two EEG graphs to determine how well the simulation was able to replicate the experimental results described earlier. The one corresponding to the default directory shows the EEG generated when weights were left at their default values. The ach EEG display is different in a number of ways. First, the amplitude of the gamma frequency oscillations is significantly attenuated. We also see high frequency activity riding on top of the remaining gamma oscillations. Although these EEG results differ significantly from what is seen in experimental preparations, the attenuation of the gamma rhythm is qualitatively similar. We may ask ourselves why the model was not able to replicate the full extent of the experimental results. Perhaps we neglected to incorporate an important parameter. For example, it is well known that acetylcholine and norepinephrine increase the excitability of pyramidal cells by blocking a slow K⁺ current (Hasselmo and Bower 1992, Madison and Nicoll 1982). Yet, neither the current nor the increase in excitability is modeled. The ambitious student may try to better match model data to experimental EEGs by incorporating these details into the model using knowledge gleaned in later chapters.

9.6 Detailed Examination of Network Behavior

So far, you have observed network behavior using only three types of data windows. As already mentioned, the interface to this tutorial actually allows the user to observe many different aspects of network response. Because the possibilities are far too numerous to describe in detail, the following sections describe some of your options and some suggested exercises. In going through the sections below, remember to specify your playback data directory as default.

If you return to the field potential button and click, beneath the title bar labeled fp's generated by synaptic currents you will find toggles corresponding to the six termini of the six different synaptic pathways going to the pyramidal cells. These toggles allow you to look at the contributions of individual synaptic pathways to the overall EEG.

By clicking on several of these at once, you can add the field potentials generated by the synaptic currents in different pathways. Thus, if you click on the toggles labeled `supIb` and `deepIb` and then click on `display` a graph will appear on the screen whose title should indicate the combination of toggles you just clicked. If you play back the simulation, you will see the summed contribution of the synaptic currents in these two layers to the overall EEG. Note that if you sum the contributions from all synaptic pathways, you will not get a graph that is identical to the EEG. This is because you are neglecting the contribution to the EEG from voltage-gated currents in the soma.

At the total network level you can observe the activities of rows of neurons all at once by selecting the button labeled `full cell view` to the right of the pyramidal cell. A menu window containing two dialogs should appear over the button. The `full cell view` option allows you to display a row of cells in the network. The parameter dialog gives you the choice of viewing membrane potential (`Vm`), transmembrane current (`Im`), or synaptic conductance (`Gk`). The `row` dialog specifies the row number to be displayed. Valid row numbers are 1 through 9. Enter `Gk` in the parameter dialog and click on `display` to see row 5 and to look at synaptic conductance. Each column you see in the display window represents a single pyramidal cell in the network. Each cell in the display is composed of six visual compartments that represent conductance changes caused by each of the six synaptic pathways terminating on the pyramidal cell. `Vm` and `Im full cell view` displays show only five visual compartments for each cell which correspond to the five electrical compartments used to model the pyramidal cell.

In addition to overall patterns of network activity, the interface allows you to examine the response properties of many individual network components. For example, if you look at the pyramidal cell in the menu window, you will notice that at the center of every cell compartment there is a button indicating laminar position. These buttons allow you to visualize network data associated with that compartment only. For example, press the button labeled `deepIb`. A menu titled `deepIb_options` should appear over the button you just pressed. Click on `network view`. Another menu titled `deepIb_network` should appear next to the original `deepIb_options` window. The `network` option allows you to display a slice of piriform cortex that is parallel to the surface, but contains only the specified compartments from all the pyramidal cells in the network. The options `Vm`, `Im`, and `synaptic conductance` indicate the parameter to be visualized. To get a better idea of what this means, click on `Vm`, `Im`, and `synaptic conductance`. Three small windows each containing a 9×15 grid of rectangles should appear on the right side of the screen. Each rectangle represents a `deepIb` compartment from a single simulated pyramidal cell found in that position within the network. The value of parameters `Vm` (membrane potential), `Im` (transmembrane current), and `synaptic conductance` are indicated by the color of the rectangles. Hot colors like red and yellow represent high values and cold colors like blue represent low values. The parameter `synaptic conductance` requires some explanation since it is not something neurophysiologists are accustomed to observing directly. Each

synaptic pathway induces a change in conductance in the compartment (or layer) on which it is acting. This change in conductance is measured as the **synaptic conductance**. Although this is not something a physiologist can yet measure, it is a useful way of keeping track of the total amount of synaptic activity. This in turn can help one to understand network dynamics. Click on **cancel** to get rid of the **deepIb_network** window and then click on the button labeled **total synaptic conductance (Na)** in the **deepIb_options** window. A graph should appear in the upper part of your screen next to the menu window. When you play back your data, this graph will show the summed synaptic conductance over the entire network for the deepIb compartment.

Other compartments contain the same visualization options and can be executed simultaneously with any combination of other compartment displays. One exercise you can perform is to determine which parameters are clearly oscillatory and which are not. For example, there is clear oscillatory behavior in the full cell view window displaying synaptic conductance (labeled **full_cell_Gk_5.default**). You should notice that the fourth row of compartments down displays a similar undulating pattern to that seen in the network display. This was to be expected since this row of compartments represents the synaptic conductance of deepIb compartments in the pyramidal cell, which is also oscillatory.

You can also use multiple output graphs to observe in detail the relationships among different components of the model. For example, you should compare the graphs for pyramidal cell spike count, EEG, supIb and deepIb field potential, and total synaptic conductance for the deepIb compartment (press the button labeled **total synaptic conductance (Na)** in the **deepIb_options** window). You will notice that all graphs display oscillatory waveforms of varying amplitudes. You might want to rescale some of the graphs by magnifying the waveforms. To do this, click on the **scale** button in any of the graph windows. Play with **ymin** and **ymax** until you have properly magnified the waveform. Do this for all the graphs making sure that **xmax** is 500 in each case. Look closely again and you will notice that there is a one-to-one correspondence between the peaks of all the graphs. This tells us a number of very important things. First it allows us to correlate the EEG to pyramidal cell activity. By comparing the graphs labeled **spike_activity_pyr_default** and **field_eeg_default** we can say that a peak in the EEG corresponds to a peak in pyramidal cell activity. To see this better, scale the EEG and spike count windows so that **xmax** is 100. Now scale the other graphs to an **xmax** of 100 and compare the EEG to the added supIb and deepIb field potentials. The peaks and valleys of the waveforms correspond almost exactly; furthermore, both waveforms have similar amplitudes. This suggests that the field potentials generated by association pathway synaptic currents underlie a significant portion of the EEG. Now compare the conductance graph to the EEG. The valleys in the EEG correspond to the peaks in the synaptic conductance. The reason for this is that the inward currents generated by deepIb synaptic conductance create a negative field potential (see Eq. 9.1). Our last comparison is between the pyramidal cell spike count and synaptic conductance. You will notice that the synaptic conductance trace is positively phase shifted with respect to the pyramidal

cell spike count. This is because of the delay between the firing of caudal pyramidal cells and the arrival of synaptic input to layer deepIb.

9.7 Summary

The piriform model has been used to study a wide range of physiological and functional phenomena including associative memory function (Wilson and Bower 1988, Wilson and Bower 1992). The model attempts to retain biological plausibility, but simplifies the structure of the piriform cortex considerably. Despite the existence of a number of cell classes in the piriform cortex, only three were modeled: superficial pyramidal cells, feedforward and feedback inhibitory cells. Pyramidal neurons are represented with five electrical compartments and inhibitory cells with only one. Although the piriform area contains on the order of millions of cells, the model (in its tutorial manifestation) uses only 405. Axon conduction velocities and anatomical circuitry were fitted to experimental data in order to place realistic constraints on the network. Despite its simplicity, the model has been able to replicate a wide range of cortical behavior and to make experimentally testable predictions. The reader is encouraged to use information in the second part of this book to modify this network model as he or she sees fit in order to further explore cerebral cortical dynamics and function.

9.8 Exercises

1. Change the synaptic weight for the feedback interneuron to pyramidal cell pathway from 1.0 to 0.0 and then run the simulation. Change the pathway back to its default weight and then change the weight for the pyramidal cell to feedback inhibitory interneuron from 1.0 to 0.0. Compare the data you obtain from each change to the default data you generated earlier. What happens to the EEG in each case? What role do the feedback inhibitory interneurons seem to play in generating 40 Hz oscillations?
2. The GABA_A receptor activates the fast ligand-gated Cl^- permeable synaptic channel responsible for the shunting inhibition generated by the feedback inhibitory cells in the layer II area of the pyramidal cell. Barbiturates are known to act at a special site on the GABA_A receptor that prolongs the burst time of the Cl^- channel (Hille 1992). Simulate the effect of applied barbiturates to the piriform cortex by changing the channel open time for the inhibitory synaptic input to layer II from 7.0 ms to 18.0 ms. What effect does this have on the EEG? What do you think is the physiological basis for this effect?

3. The olfactory bulb is known to generate both fast and slow EEG rhythms that match those found in the piriform cortex when recordings are done simultaneously (Bressler 1987). In the piriform model, bulb input is modeled by a random number generator with a flat frequency distribution; nonetheless, fast oscillations still occur at roughly 40 Hz. Change the synaptic weight of the afferent pathway (coming from the bulb via the LOT and going to the pyramidal cell) to 0.1. Then generate a new batch of data where the synaptic weight of the afferent pathway is set to 5.0. Compare the EEGs you just generated to the default case. Is there any significant difference? Do you think piriform cortex requires patterned stimulation from other brain areas in order to generate 40 Hz oscillations? If not, what purpose might patterned input serve?

Chapter 10

The Network Within: Signaling Pathways

UPINDER S. BHALLA

10.1 Introduction

In the preceding chapters, we have taken the building blocks for neuronal models in the form of ion channels, membrane compartments and synapses (Chapters 4, 5 and 6) and put them together to form neurons, small neural circuits, and then networks (Chapters 7, 8 and 9). We conclude this section of the book by opening the lid on the building blocks, and seeing what happens inside them. There used to be a view of the atom as a tiny solar system, which led to exotic visions of an infinite series of ever-decreasing worlds as one examined matter at finer and finer scales. Neurobiology now has to face a similar unsettling concept: that hidden within “atomic” compartments and under all the ion channels, there exists a whole new universe of subcellular networks. These networks, of course, are the multitudes of interacting biochemical signaling pathways. They profoundly affect everything from the properties of single channels to the morphology of neurons and the wiring of the brain itself. Recent work on biochemical signaling reactions has emphasized the sheer complexity of these networks. There are dozens of known major signaling pathways, each having at least five to ten enzyme isoforms, each of which communicates with a different subset of other messengers and pathways. As with neuronal networks themselves, these biochemical networks cannot be analyzed in isolation. Neuronal signaling events from above, and nuclear and metabolic events from below, provide a barrage of signals that this network must process.

What role do signaling pathways play in the life of a cell? In a word: everything. Metabolism, cytoskeleton formation, gene regulation, response to external inputs, differentiation, cell cycle, synaptic computation — all these operations are in the domain of signaling pathways. Consider a large industrial chemical complex. The control system for such a system is typically a huge computer, with hundreds of control points and sensors. A living cell is a chemical system whose complexity is orders of magnitude beyond anything man-made, and its control systems are correspondingly complex. Although a lot of the “software” is encoded in DNA, the role of “hardware” (more properly, “wetware”) is carried out by biochemical pathways.

The goal of this chapter is to introduce you to biochemical computation, with particular emphasis on aspects important for neurobiology. These include gating and modulation of ion channels, signaling by diffusible and retrograde messengers, and the myriad processes involved in long term potentiation (LTP). After a brief introduction to a few major pathways, the chapter covers some of the theory underlying biochemical models. The second half of the chapter describes the specifics of building such models using GENESIS and *Kinetikit*.

10.1.1 Nomenclature

A *signaling pathway*, in the sense I employ it, is any series of reactions that manipulate information of importance to a cell. This definition is rather broad, and might, for instance, be considered to include metabolic regulatory pathways that do not have much to do with the outside world. We are mostly concerned with pathways that do involve interaction with external events, particularly neuronal events.

It is sometimes difficult to decide where one pathway ends and another begins. I will use the term *pathway* to describe a group of reactions influencing a single signaling enzyme. In this sense, then, the classical cyclic AMP pathway shown in Fig. 10.1 (Gilman 1987) consists of at least four pathways: the receptor-ligand pathway, the G protein activation pathway, the adenylyl cyclase pathway, and the protein kinase A activation pathway. Another way of looking at it is to consider a pathway as a node at which information can converge, be processed, and sent on to multiple other nodes.

The term *second messenger* is typically used for a small signaling molecule (i.e., not a protein) that is produced indirectly through the action of some primary messenger such as a neurotransmitter. Examples are cAMP (cyclic adenosine monophosphate), DAG (diacylglycerol), IP₃ (inositol 1,4,5 triphosphate), AA (arachidonic acid), and, of course, Ca (calcium). Given the proliferation of interactions among signaling pathways, it is sometimes difficult to decide if a given molecule is a second, third, or *n*th messenger. We will apply the term to most small non-protein signaling molecules.

The term *signal transduction* will be applied in a restrictive manner to receptors that change or transduce one signaling modality (such as light) to another, such as biochemical

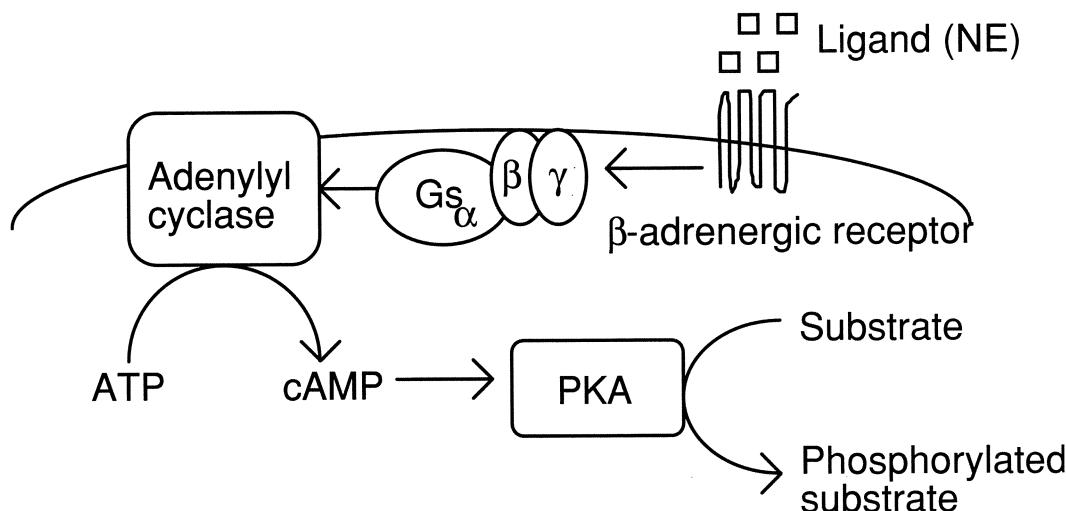


Figure 10.1 The cyclic AMP pathway. In this chapter we treat it as four successive pathways: (1) activation of the β -adrenergic receptor, (2) activation of the G protein G_s, (3) activation of adenylyl cyclase, and (4) activation of protein kinase A. This pathway was one of the first to be examined in detail, and incorporates many of the interactions subsequently found in other pathways.

signals. In this sense a Ca channel is a sort of signal transduction mechanism, but the more specific term *ion channel* will be used in this case. *Receptors* will be used to describe proteins that detect neurotransmitters, usually at a synapse.

10.1.2 A Short Short Course in Biochemistry

This little section is meant for non-biologists and should be skipped by anyone who has survived a modern biology course. Any of the standard textbooks on biochemistry or molecular biology of the cell are recommended reading (e.g., Alberts, Bray, Lewis, Raff, Roberts and Watson 1994, Kandel, Schwartz and Jessel 1991).

DNA (deoxyribonucleic acid) is the genetic material of the cell. It encodes information in the sequence of four chemical building blocks called *bases*. These are labeled A, T, C and G.

A *protein* is a chain of amino acids (another building block). There are 20 amino acids. The sequence of amino acids is specified by DNA, and in turn determines the three-dimensional structure and biochemical properties of the protein. Proteins frequently associate in pairs to form *dimers*, or triplets (*trimers*), or larger numbers (*multimers*).

An *isoform* of a protein is another protein, with a different but usually closely related sequence. Isoforms usually have similar enzymatic properties, but may be regulated differently.

An *enzyme* is a protein with catalytic activity. It speeds up reactions. Enzymes usually exhibit great specificity in their *substrates*, the molecules they act upon, and also in the *products* they generate.

ATP (adenosine triphosphate) is the major source of chemical energy in the cell. The removal of one or two phosphates from ATP releases energy. This reaction is called *hydrolysis*. GTP (guanosine triphosphate) is a chemical cousin of ATP, and shares its energetic properties but is not involved in as many reactions in the cell.

A *protein kinase* is an enzyme that adds a phosphate from ATP to an amino acid on a protein. This process is called *phosphorylation*. The added phosphates frequently modify the activity of the target protein, which makes this a process of great importance for signaling.

A *phosphatase* reverses the action of a kinase: it removes the phosphate group.

Buffering is a way of holding the free concentration of a given molecule close to a desired set point by a suitable combination of chemical sources and sinks for the molecule. It is most commonly used for holding the *pH*, that is, the acidity, of a solution at a desired level.

10.1.3 Common Signaling Pathways

Regrettably, this section looks rather like an alphabet soup. It can be skimmed through on a first reading, but it is actually the barest of introductions to the topic. Anyone contemplating research simulations on signaling will need to go into much greater breadth and depth than this incomplete list. You should assume that there are tens of known examples in each category, each of which has tens of known isoforms. You should also be aware that the field is in an exponential growth phase: new isoforms are reported almost daily, and entirely new pathways and signaling mechanisms turn up every few months.

Receptors

These collect information from outside the cell, and couple it into intracellular pathways. There are two main subclasses of receptors. Ligand-gated ion channels are directly gated by receptors such as the familiar NMDA receptor, and pass signaling ions such as calcium into the cell when suitably stimulated by the presence of a ligand. G protein coupled receptors, such as the beta-adrenergic receptor, belong to a second category that gates channels indirectly. These receptors activate G proteins when stimulated by ligands. The G protein may then either modulate channel activity by binding directly to the channel, or regulate the activity of an enzyme involved in a second messenger pathway that modulates channel activity.

G Protein Pathways

The term *G protein* refers to proteins that bind guanine nucleotides, such as GTP. There are at least four different major G protein families, G_s , G_i , G_o , and G_q . G_t (transducin), in the visual receptor pathway, is a G protein in the G_i family that is stimulated by light. As usual, each family has five to ten known isoforms. These trimeric, membrane-associated signaling proteins contain α , β , and γ subunits. The β and γ subunits have their own sets of isoforms, so there are plenty of permutations. Only a small fraction of these are believed to occur in nature. Activated α wanders around on its own, activating further pathways, but the $\beta\gamma$ complex remains dimerized. The prototypical G protein reaction is shown in Fig. 10.2.

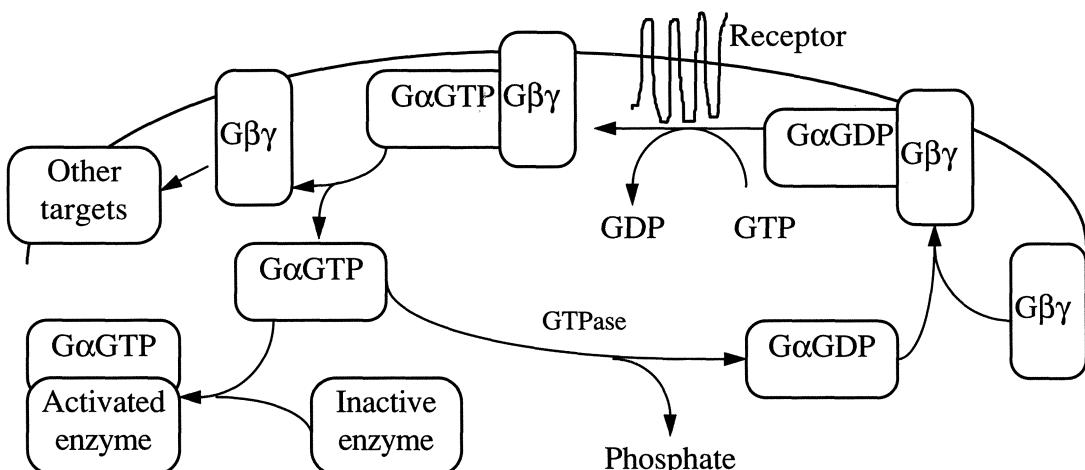


Figure 10.2 G protein signaling cycle.

The main steps in the cycle are as follows. First, the ligand activates the G protein coupled receptor. The activated receptor promotes the replacement of GDP by GTP, following which, the active G α -GTP splits off from the G $\beta\gamma$ dimer. The active G α binds to an effector enzyme such as adenylyl cyclase, and turns it on. In parallel, G $\beta\gamma$ binds to its own target enzymes or channels. After a while (a few seconds to minutes), the slow GTPase activity of the G α hydrolyzes the GTP to GDP. Now the pieces reassemble: G α -GDP and the G $\beta\gamma$ associate, and the inactive heterotrimer can bind to another receptor to repeat the cycle. There are two major features to note here. First, the G protein loop *amplifies* signals. A receptor can catalyze the activation of numerous G proteins while the ligand is bound. Each G α remains active until the GTP is hydrolyzed, and this takes a minute or so. During this time, the activated effector enzyme can process a large amount of substrate. Second, there are *two* signaling outputs from a G protein: the G α and the G $\beta\gamma$. Each can potentially influence several signaling pathways.

There is also a large subfamily of “small G proteins,” which lack $\beta\gamma$ subunits. These

include Ras, Rab, and many others. They have their own family of regulatory proteins that affect the rate of GTP turnover.

Protein Kinase Pathways

Protein kinases add phosphate groups to specific sites on other proteins, and thereby affect their activity. There are dozens of kinases known, which fall into two main classes: serine/threonine kinases and tyrosine kinases. These classes indicate the amino acids that are phosphorylated by the kinase. The major serine/threonine kinases are PKC (protein kinase C), PKA (protein kinase A), CaM-KII (calcium-calmodulin regulated type II kinase), and MAPK (mitogen-activated protein kinase). Receptor tyrosine kinases (RTKs) are an important class of tyrosine kinase. Beyond the specificity for a particular amino acid substrate, each kinase has its own range of sequence preferences, which can be exquisitely specific or very broad. Kinase activity is regulated in many ways, including second messengers such as Ca, DAG or cAMP, phosphorylation, membrane association, or even direct ligand binding.

Phosphatases

These reverse the activity of kinases: they remove phosphate groups. Like kinases, they have their own classes, isoforms, substrate specificities and so on. Major phosphatases are PP-2A (protein phosphatase 2 A), PP-1 (protein phosphatase 1) and calcineurin. It used to be thought that their only role was to balance out kinase function, but it is now clear that they are in turn regulated in a wide variety of ways and contribute actively to signaling.

Phospholipases

These take phospholipids, which are a major constituent of the cell membrane, and turn them into active signaling molecules. Examples include PLA₂ (phospholipase A₂), PLC- β (phospholipase C- β), and phospholipase D. PLC- β is particularly interesting as it produces two signals: diacylglycerol (DAG) and IP₃ (inositol triphosphate). PLA₂ produces AA, which is a membrane-permeable signal and may be involved in retrograde signaling. It should be pointed out that many phospholipids themselves are important in signaling.

Cyclases

The main cyclases are AC and GC (adenylyl and guanylyl cyclases), which produce the cyclic nucleotides cAMP and cGMP from ATP and GTP, respectively. There are at least ten ACs known, each of which has its own regulatory preferences. All ACs are activated by G_{s α} ; most are inhibited by G_{i α} , and there is regulation by $\beta\gamma$, phosphorylation, and so on, in various combinations.

Phosphodiesterases

These degrade cAMP and cGMP, and thereby turn off signals from cyclases. Regulatory mechanisms include CaM-dependence and phosphorylation.

Calcium

Although calcium is just a simple ion, and not really a pathway, it is arguably the most important single signaling molecule. The regulation of Ca alone involves ligand and voltage-gated channels, pumps, intracellular stores, buffers, diffusive microenvironments and more. It is involved in the regulation of almost all kinds of pathways, and there are whole families of proteins such as CaM (calmodulin), whose only role is to detect Ca levels as part of the regulation of other proteins.

10.2 Modeling Signaling Pathways

10.2.1 Theory

Chemical rate theory is an old, well-established subject. A lot of it has to do with analytical derivation of results that we, in our modern-day decadence, leave to computers. For our purposes all we have to understand is a single rate equation:

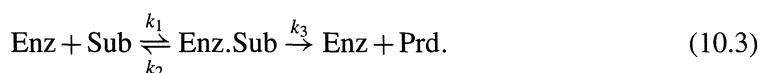


By definition, this is described by a differential equation of the form

$$\frac{d[A]}{dt} = k_b[C][D] - k_f[A][B], \quad (10.2)$$

where the square brackets indicate the concentration of the specified molecule. The mass conservation laws constrain the remaining concentrations, once the starting concentrations and the current value of [A] are known. You have already encountered very similar rate equations when studying the Hodgkin–Huxley model of activation of sodium channels (Eqs. 4.11–4.13).

Enzyme catalysis is a very important part of biochemical reactions. One of the simplest and best descriptions of enzyme kinetics is due to Michaelis and Menten. Way back in 1913, they described catalysis as a two-step process, where the enzyme and substrate first reversibly combine to form a complex (Michaelis and Menten 1913). This complex can then go forward to form the product in an essentially irreversible step:



Although we need three rate constants here, Michaelis and Menten went on to specify only two parameters which do an excellent job of representing enzyme properties. These are:

$$K_m = (k_2 + k_3)/k_1 \quad (10.4)$$

and

$$V_{max} = k_3. \quad (10.5)$$

K_m is the Michaelis-Menten constant for the enzyme. It is the concentration of substrate at which the rate of generation of product is half maximal. I leave the derivation of this as an exercise for the reader.

V_{max} is the maximum velocity of the enzyme, i.e., the rate of formation of product when there are saturating amounts of substrate present. All of the enzyme will then be complexed with the substrate, so V_{max} is just k_3 .

10.2.2 Sources of Data

A brief digression on sources of data is in order here. It is almost certain that a modeler in search of biochemical data will have to refer to the original literature, both for the reaction mechanisms and for the parameters. *Journal of Biological Chemistry* accounts for about half of the work on the subject. It provides its entire contents on the Internet (<http://www-jbc.stanford.edu/jbc/>) and on CD-ROM, making literature searches almost pleasant. The remainder is partitioned between many other journals, such as *European Journal of Biochemistry*, *Proceedings of the National Academy of Science*, *Biochemistry*, and many others.

10.2.3 Figuring Out the Mechanisms

Signaling pathways are typically represented as neat little black boxes connected with arrows. The first step in constructing a model of such a pathway is to open up the black box and recoil in horror at the seething mass of interactions inside. These interactions, which are frequently ill-defined in mechanistic terms, must be framed in terms of individual chemical reactions. In GENESIS, these are binding/reversible reactions, and Michaelis-Menten enzyme reactions. Complete mechanistic details for signaling pathways are available in only a few cases. G protein signaling, for example, has been worked out in considerable detail (Fig. 10.2, and Gilman 1987). The classical pathway for cAMP signaling has also been extensively studied and modeled (Levitzki 1984). Lauffenburger and Linderman (1993) provide many examples of signaling pathways that have been modeled with varying degrees of realism. It is much more common to have partial mechanistic details available. For example, several important kinases including PKC and PKA are known to incorporate an enzyme site and a pseudosubstrate region, which binds to and inactivates the enzyme site.

The process of activation of these enzymes can then be described in terms of the release of the pseudosubstrate from the enzyme site. As a modeler, one may have to make compromises on several fronts:

- If one is lucky, the mechanistic data may actually go beyond what you need or can handle. PKA regulation is a case where there is such an overabundance of information (e.g., Døskeland and Øgreid 1984). Judicious simplification is usually not difficult in such cases.
- At the other extreme, mechanistic details may be so sparse that you have to develop an “empirical” model, which uses the simplest possible mechanisms that fit the observed data. This approach works surprisingly well, especially if there are plenty of data to constrain the system. A common situation is where there are several known activators for an enzyme, each of which is described by a different concentration-effect curve. The dumbest approach is then to treat each activation process as a separate reaction, resulting in an activated enzyme with different rates. This has been done, for example, for the activation of PLA₂. One can often go a step further and rule out certain possible mechanisms on the basis of such raw concentration-effect curves. This is discussed below.
- The general, and most common case, is that some of the crucial mechanistic details are known, and the rest are a bit fuzzy. One can often fall back on mechanisms for similar pathways to fill in the gaps. Otherwise, one just has to plug the holes with the simplest mechanisms that will work.

As an example of figuring out mechanistic details, consider protein kinase C (Nishizuka 1992). It is known to be regulated by a pseudosubstrate domain that normally binds to and blocks the kinase site. Activators function by releasing this blockage, in assorted ways. This crucial regulatory information enables us to predict that most activators will expose an enzyme site of identical activity. This implies that the strength of the activator is likely to depend on its efficacy in releasing the block, rather than on special properties of the enzyme-activator complex. So, the active enzyme can be represented as a single pool, to which different activators contribute an amount determined by their respective efficacy. To round off the picture, let us consider activation by Ca, DAG, and AA. This lets us draw up the first skeleton of the activation process (Fig. 10.3). We just need to sum up the individual contributions into a total final activity, as we assume that the exposed enzyme site is identical in all cases. We can further elaborate on the activation process by noting that each of the activators works synergistically with the others. One way of modeling this might be to add further reactions where the two activators combine, as illustrated by the Ca-AA synergistic pathway in dotted lines. As it turns out, we can considerably refine the activation mechanisms because the concentration-effect curves require even more complex interactions for the model to fit well.

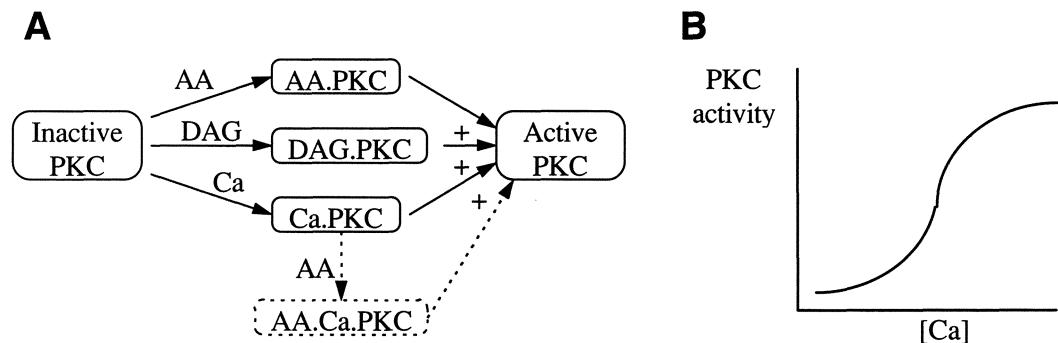


Figure 10.3 (A) Skeleton model of PKC activation. (B) Example concentration-effect curve.

10.2.4 Reaction Rate Constants

When a signaling pathway is modeled in terms of biochemical reactions, the immediate parameters required are rate constants and initial concentrations. Unfortunately, biochemists rarely provide parameters with modelers in mind. The most common form for expressing biochemical data is the *concentration-effect curve* (Fig. 10.3B) which describes an *effect*, such as activation, rate of production, or some similar experimental quantity, in terms of the *concentration* of an activating agent. Mechanistically, the link between the concentration and the effect is quite likely to involve numerous intermediate steps. By the time you get to the point where you want to fill in rates, we assume that you have a specific mechanistic model in mind, so you should have an idea of what these steps are. Obviously, you want to start with the simplest curve involving the fewest unknowns. If one must express a concentration-effect curve as a single number (losing information in the process) it is the concentration at half-maximum, K_d . Assuming that your reaction is first-order, $K_d = k_b/k_f$. If in addition one has information about the time-course τ of the reaction, one can completely define k_b and k_f . This procedure is fine in theory. In practice, if you do not actually run your simulation and compare the concentration-effect plot point-by-point, you will throw away a lot of valuable data and probably lose the chance to catch serious mistakes in your assumptions. Many papers report families of concentration-effect curves under various conditions. Such sets of curves embody enormous amounts of information to help constrain rates, mechanisms, and interactions between different activators.

10.2.5 Enzyme Rate Constants

Enzyme rate constants are usually reported as the maximal enzymatic rate V_{max} and the Michaelis-Menten constant K_m . This is obviously insufficient data for the three rate constants in the standard formulation for an enzyme reaction. As already mentioned, the maximum velocity of the reaction $V_{max} = k_3$. From the Michaelis-Menten definition, $K_m = (k_3 +$

$k_2)/k_1$. One option to fill in the missing rate constant is to assume that k_3 and k_2 are in a fixed ratio:

$$k_2 = x k_3. \quad (10.6)$$

Then,

$$k_1 = (1 + x)k_3/K_m. \quad (10.7)$$

I use a value of $x = 4$. Exploring a huge range of such scale factors (from 0.4 to 40) leads me to believe that the behavior of most models is remarkably insensitive to the exact value of x . In other words, the Michaelis-Menten formulation captures most of the essential properties of the enzyme.

There are some exotic complications to consider when obtaining enzyme rates from the literature. Many membrane-bound measurements are conducted in artificial membrane systems. The composition of these membranes may strongly affect rates. One also has to be extremely careful of situations where the enzyme has access to only a limited amount of substrate. This may happen in experiments where both enzyme and substrate are micelle-bound, for example, PLA₂.

10.2.6 Initial Concentrations

Determining initial concentrations for signaling molecules is a task fraught with peril. In principle, one just has to look up purification stages for enzymes, and standard biochemical measurements for substrates and messengers. In practice, one needs to watch out for:

- Tissue and species specificity
- Partitioning between different fractions of tissue (membrane, cytosolic, particulate, etc.)
- Purification methods, and yield and loss of activity during purification
- Loss of activity (or sometimes even gain of activity!) during storage

and many other complications. Given all these hazards, the more references for each concentration, the better.

Typically, one will need to provide only a few initial concentrations and just let the simulation run to steady-state to obtain the remainder. This is essential, as it is experimentally very difficult to obtain steady-state values for many of the reaction intermediates. If one has equilibrium values for some molecules, that is a bonus and an excellent cross-check.

10.2.7 Refining the Model

There is no final step to building a good model of any kind. In signaling models in particular, it takes many iterations to converge to a good representation of the system. All the preceding steps, (except hopefully the theory) will need to be revisited as the model evolves. Most of my models of individual pathways have involved 20 or more cycles of improvement. An important part of this refinement includes repeated scans through the literature to hunt out alternative sources for each parameter, and cross-checks that become feasible as the model becomes more predictive.

10.3 Building Kinetics Models with GENESIS and *Kinetikit*

As with other simulations and demonstrations in this book, we draw a distinction between the underlying GENESIS simulation modules and the user interface based on them. The modules for kinetic modeling are provided by the kinetics library, whereas *Kinetikit* (or *kkit*) is the user interface. *Kinetikit* is similar to *Neurokit* in the sense that it is a research tool rather than a tutorial program. It makes much better use of the features provided by XODUS, and we hope you will find it easier to use.

10.3.1 The Kinetics Library

As you will learn in Chapter 12, a GENESIS library contains the definitions of commands that will be accepted by GENESIS, as well as basic building blocks, called *objects*, which are used for the construction of simulations. The kinetics library defines additional commands and objects to extend the capabilities of GENESIS to simple models of biochemical signaling. By “simple” I merely mean that the only interactions considered are chemical. There is ample complexity in biological signaling, even when such vital complications as diffusion, compartmentalization, and enzymatic scaffolding are not addressed.

A simple exponential Euler scheme (Sec. 20.3) is used by the kinetics library. Fortunately, stiffness does not seem to be such a problem in the kinetic computations. Time steps of 2 msec are sufficient for most simulations of biochemical reaction systems. It is trivial to implement a crude variable time step scheme when known steep stimuli occur — just lower the time step. About 100 μ sec will usually give you sufficient stability and accuracy for such cases. A **ksolve** object (cousin to the **hsolve**) is being designed to use much faster implicit and variable time step techniques. Even the current methods work many times faster than real-time for all but the most complex simulations.

All calculations are carried out on numbers of molecules, rather than concentrations. This simplifies computations where molecules are exchanged between chemical compartments of different volumes. However, concentrations are calculated locally by each molecular pool, by dividing the number of molecules by a *vol* field of the **pool** object. The *vol* field is typically

scaled to include Avogadro's number so that the concentration units are μM or some other familiar units. The kinetics library assumes bulk quantities of all molecules. Markov models are specifically not simulated in the current kinetics library. Caution should therefore be employed when dealing with tiny compartments: it has been estimated that there are about ten Ca ions in a small dendritic spine. Mean rate theory may not apply in such situations.

The GENESIS objects provided by the kinetics library are the **pool** of molecules, the **reac** for simple reactions, and an **enz** which can be attached to a reactant pool to provide an enzyme site. A simple **concchan** object is provided for constructing concentration-driven channels within the kinetics library. It is a much simpler version of the voltage- and ligand-gated channels used elsewhere. These objects are described in much more detail in the GENESIS Reference Manual.

10.3.2 Kinetikit

The method of choice for developing kinetic simulations in GENESIS is to use *Kinetikit*, which is a graphical interface and simulation tool designed specifically to make it easy to build and manage complex kinetic simulations. *Kinetikit* is internally documented in detail. To get you started quickly, we'll go through a demonstration simulation that illustrates many of its features. This demonstration involves a simple reaction where a molecule A reversibly converts into a molecule B:



Step 0

Make sure you have a version of GENESIS that includes the kinetics library. If there is a startup message of the form “The kinetics library is copylefted under the LGPL, see kinetics/COPYRIGHT.” then it is there. If not, you will need to follow the installation steps detailed in the kinetics documentation. Once you have it installed, you are ready to go. After changing to the directory in which *Kinetikit* resides (usually *Scripts/kinetikit*), perform the following steps.

Step 1

Type:

```
genesis kkit
```

This will display a start-up window detailing the philosophy behind *Kinetikit* models, and also indicating the licensing terms (free!). As soon as *Kinetikit* has loaded, this window will vanish and will be replaced by a screen similar to that in Fig. 10.4. The control window

includes various menu options on the top. The simulation run control buttons in bright traffic-light colors are just below, and dialogs for the simulation duration and current time below them. At the bottom is a row of strange icons. These are the building blocks for a kinetics simulation, and their role will shortly become clear. Below the control window is the *Kinetikit* edit window, where the reactions are set up. It will initially be blank. The graph windows are the only other windows currently visible.

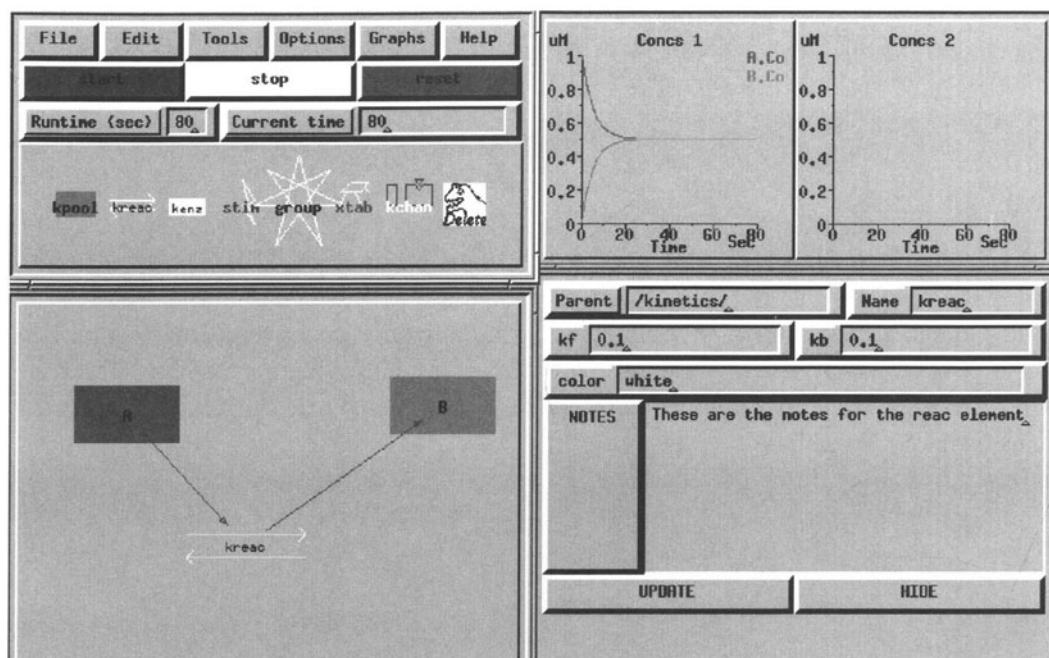


Figure 10.4 Screen dump of *Kinetikit*.

Step 2

Use the left mouse button to click and drag the first of the icons, the blue kpool icon, into the edit window. Two things should happen: a kpool icon should appear in the edit window, and a new window should pop up with lots of dialogs for parameters. What you have just done is to create a new pool of molecules in the simulation, and to help you along, the parameter window has been displayed. Typically the first thing one does is to change the color and name of the icon to something that suits you. Try red and A. Then hit the HIDE button in the parameter window.

Step 3

Familiarize yourself with the properties of the edit window. Move the kpool icon in the edit window around a little, using click-and-drag with the mouse. The layout of the reactions has nothing to do with the numerical calculations, but a clean layout makes it much easier to figure out what is going on. Now double-click on the kpool icon. The parameter window reappears. Set the initial concentration CoInit to 1. Now drag the kpool icon to the graph window. You have just made yourself a plot, labeled A.Co. The “Co” suffix indicates the field that is being plotted, in this case the concentration Co of element A.

Step 4

Drag in a reac and another kpool (let’s call it B). Add the B to the graph as well. Now we can set up a reaction. Drag A onto reac, and then reac onto B. Little green arrows should appear, linking them. If your reaction components are too close together, the little arrows don’t fit, and so they vanish. You can zoom and pan the edit window using the angle-bracket and arrow keys, to see this effect. Also try rearranging the reaction using click-and-drag within the edit window.

Step 5

Run the reaction. Hit the green start button, and watch the reaction proceed. You can hit the stop button in mid-stream, and start up again without affecting the calculations. The reset button clears the simulation. Play around with the parameters of the pools (especially CoInit, nInit and vol), and the reaction (kf and kb) to get a feel for how it all works. Double-click on A or B while the simulation is running, or hit the UPDATE button in the parameter window, to monitor concentrations. Also try double-clicking on the graph axes and on the plot labels, to pop up parameter windows for specifying display parameters.

Step 6

Edit the reaction. Drag A onto kreac again. The green arrow disappears. Repeat the drag, and it reappears. This is fine for simple editing, but what happens if you want to make it a second-order reaction $2A \rightleftharpoons B$? You will have to go to the Options menu and enable higher-order reactions to accomplish this feat. Now drag kreac onto Barney, the evil dinosaur icon in the control window. Chomp! Barney has deleted kreac. This works for everything in the edit window, and also for plots.

Step 7

Save the reaction. Click on the File menu button, fill in some notes, and enter your chosen filename, e.g., “`reacs.g`.” It will be a script file, so you should have the usual “`.g`” suffix. Having saved the file, you can restore your simulation by typing:

```
genesis reacs.g
```

I will assume, for the remainder of the chapter, that you have the good sense to save each version of your simulations frequently.

Exercise for the reader:

Make and test an enzyme reaction. Hint: an enzyme site cannot exist without an enzyme. So, you will need to drag the enzyme onto an existing pool.

10.3.3 A Feedback Model

As a more complete example of the development of a kinetics simulation, we will use *Kinetikit* to come up with a feedback pathway that exhibits some interesting properties such as bistability. We then show how to run this model without the interface, and finally how to hook this up to other GENESIS components.

The Model

The feedback model consists of two enzymes, X and Y, each of which is activated by the product of the other enzyme. The activation process is second-order. We assume that the substrates are buffered: their concentration is fixed no matter how much is used up. The products are degraded in a simple first-order reaction to avoid runaway feedback. The reactions are represented in Fig. 10.5. The figure has almost a one-to-one relationship with the *Kinetikit* implementation, so it should be easy to implement. The saved model is available in the *kinetikit/examples* directory as *feedback.g*.

Notes on setting up the feedback model

1. All enzyme and reaction rates are 1.0, except for the degradation reactions, which have a k_f of 0.16 and a k_b of 0. Note that these are *not* the default values, so you will have to explicitly set them.
2. All initial concentrations are 0.0, except the X and Y substrates, which are buffered to an initial value of 1, and the inactive forms of X and Y, which are at an initial value of 1 but are not buffered.

3. Two molecules of product bind to one molecule of inactive enzyme to activate it. Look up the Options menu to see how to do this sort of second- and higher-order reaction. All other reactions are first-order.
4. The arrows with a double bend represent enzyme reactions.
5. The default time step of 0.01 is good for running the model.
6. Plot out the concentrations of X and Y to monitor the simulation. You can also double-click X or Y at any time to read out the concentration from the Co dialog.
7. Do not use spaces in the names of the components in the model. It will confuse the simulator when it tries to restore simulations from a file.

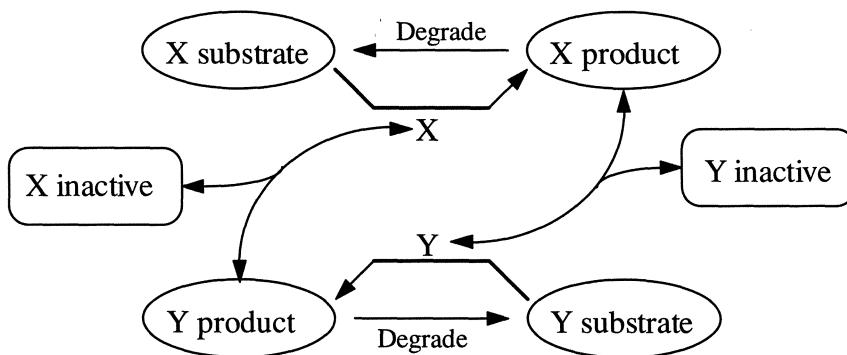


Figure 10.5 Reaction mechanism for feedback model.

Bistability

As a preliminary check on your model, just run it for, say, 100 seconds. X and Y should remain steady at a concentration of 0.0. The steady-state value you have now reached is the lower bistable point. To convince yourself that there is an upper steady-state mode, dump some X product into the system by setting its Co to 2, using the dialog box that will appear when you click on X_prd. Watch the system climb up to the upper activated state. Let it run a while longer, and convince yourself that the system has now stabilized at a new level. Note this level. If you are ambitious, you can now figure out how to push the system back and forth between the two stable levels. Are they always the same?

Concentration-Effect Curves

This manual fiddling with reaction parameters is a rather cumbersome way of identifying a bistable system, and does not easily generalize to a complex model. There is a simpler,

experimentally more feasible, and theoretically informative way of characterizing a biological feedback system of this kind. This technique works by “breaking” the loop at some point, and plotting concentration-effect curves of X vs. Y and vice versa. When the two curves are plotted on the same axes, so that the X vs. Y curve now has been flipped around, their intersection points define the behavior of the system. You can convince yourself that a single intersection point always defines the steady-state, and that if you have three intersection points the outer two must be steady-state and the middle one the threshold. This is discussed in more detail in Bhalla and Iyengar (1997). We will use two methods for generating the concentration-effect curves. In each case, we buffer one of the enzymes to break the loop, and step it through a series of concentration values.

Brute force The simple way of doing this is to double-click on X, and flip the `buffer` toggle on. Now the concentration of X is set by `CoInit`. Run until it reaches steady-state, and note the concentration [Y]. Increase X, and repeat until Y saturates. Now go through the whole process again, with Y buffered and X free. Hmm . . . The numbers look familiar. Could you have predicted this?

Using xtab If you have to generate the concentration-effect curve more than once, say, for different parameters, you will probably wish to automate the whole process. The `xtab` object is good for this. This is an extended object (see Sec. 14.5) which is created by *Kinetikit*. Drag in an `xtab` icon, and connect it up to X. Set up the `xtab` waveform to a “staircase” of levels, with appropriate times for each step. You may need to refer to the *Kinetikit* documentation to help you with this. Activate the `xtab`, and run the simulation again. Your concentration-effect curve will be generated without further effort on your part.

Exercises:

Using the numbers you painstakingly noted down before, plot out the two concentration-effect curves of X vs. Y and Y vs. X. Put them both on the same axis and find the intersection points. You may need to smooth the curves, or find extra points. Do they tally with your previous estimates for the bistable points? Devise a way of testing the accuracy of the estimate of the threshold. Hint: adjust the degradation rates again to set up an initial condition for X and Y. Alternatively, buffer the product concentrations of X or Y. Why won’t it work if you set up the initial condition by buffering X or Y to the initial desired level, and then release the buffering?

This model is rather unrealistic because the baseline activity of X and Y is zero. You could put in a baseline activity by setting k_b of the degradation pathway to some non-zero number. Find a number that gives reasonable results.

10.3.4 Beyond Kinetikit

The simulation files generated by *Kinetikit* are regular GENESIS script files. This means that you can manipulate them and hook them up to simulations in much the same way as other script files. (You will best understand this section after reading the introduction to GENESIS programming in Chapter 12.)

Batch Mode

The best part of a graphical interface is often the ability to turn it off. In *Kinetikit* this, and many other simulation defaults, can be assigned through the *PARMS.g* file. Make a copy of the *PARMS.g* file in your current directory, and edit it to set the *D0_X* flag to zero. Now when you load your simulation, there will be a few minor complaints which you can ignore. The simulation can be examined as usual through the command line, but no XODUS modules will be displayed. This is an ideal arrangement for those long overnight runs that you want to grind away in the background.

Modifying Parameters

The most common application for batch mode runs is parameter searches. Modifying the parameters in such situations is just a matter of using standard *setfield* and *getfield* commands, usually in batch files. When you combine this with the standard GENESIS script commands for saving values to data files, you have an effective way of automating long runs. For example, suppose we wished to run our concentration-effect curves for the feedback loop with various rates for the degradation pathways. This could be accomplished using the following script file.

```
//genesis
include fb.g // This is the file with the feedback model.
// We assume that the enzyme X is in /kinetics/X; and Y is in /kinetics/Y.

// This function generates the conc-eff curve for the enzyme enz,
function vary_enz(enz)
    str enz
    float conc
    float origconc = {getfield {enz} CoInit}
    float dconc = 0.1
    setfield {enz} slave_enable 4 // buffers the enz to CoInit
    echo % varying enz = {enz} >> conc_eff_file
    reset
    for (conc = 0.0; conc < 2; conc = conc + dconc)
        setfield {enz} CoInit {conc}
        step 200 -t
```

```

echo {getfield /kinetics/X Co} {getfield /kinetics/Y Co} \
      >> conc_eff_file
end
setfield {enz} \
    slave_enable 0 \
    CoInit {origconc}
end

float dkf = 0.02
float kfx, kfy
// This loop varies kf for the X and Y degradation pathways, and generates
// a concentration-effect curve for each case.
for (kfx = 0.08; kfx < 0.24; kfx = kfx + dkf)
    setfield /kinetics/degrade_X kf {kfx}
    for (kfy = 0.08; kfy < 0.24; kfy = kfy + dkf)
        setfield /kinetics/degrade_Y kf {kfy}
        echo % kfx={kfx}, kfy={kfy} >> conc_eff_file
        vary_enz /kinetics/X
        vary_enz /kinetics/Y
    end
end
quit

```

The output of such a run would be a set of concentration-effect curves that could be examined to find all the intersection points.

Saving Outputs

There are various ways you can save results of simulations in *Kinetikit*. The simplest is to create plots for the desired quantities, run the simulation, and dump the results from the plots. This can be done for individual plots (double-click on the plot) or for all of them (go to the Graphs menu). The example script above illustrates another way of saving specific quantities to a file. The standard GENESIS commands for dumping figures to postscript files (by typing Ctrl-p within the window) also work for the graph and edit window. There are special postscript options in the Tools menu.

10.3.5 Connecting Kinetic Models to the Rest of GENESIS

Kinetikit allows you to do a lot of things, but only with a few kinds of GENESIS building blocks. It is often desirable to link *Kinetikit* models to the rest of GENESIS, especially to develop models that span multiple levels of neuronal function. Again, one can take advantage of the fact that *Kinetikit* models are stored in regular script files. There are two main ways of hooking kinetic models up to the rest of GENESIS.

1. Tabs A very efficient way of coupling different kinds of simulation is to use tables to provide inputs to *Kinetikit* or to a batch file based on a kinetic model. For example, you might want to feed the Ca levels from a single cell model into a kinetics simulation. Aside from its simplicity and computational efficiency, the advantage here is that the time steps used in the two situations can be radically different. The big drawback of this approach is, of course, that it assumes that the source of the tabulated data doesn't care what the kinetic part of the model does.

2. Direct messaging This is the “proper” way of hooking up kinetic and other simulations. The simplest way to get information into kinetic models is to identify a molecule whose concentration is determined by the non-kinetic part of the model, and that is explicitly modeled there. Consider Ca influx through an ion channel, which may be modeled using a **Ca.conc** object. We just need to create a kinetic “slave” counterpart of this **Ca.conc**, and hook it into the kinetic model. Likewise, messages can be sent from “pools” from the kinetics library to specify concentrations used by other GENESIS modules.

It is often necessary to do unit conversions in such situations. For example, we may need to factor in Faraday’s constant, or Avogadro’s number, or the volume of different cellular compartments. The **xtab** module in *Kinetikit* (which is based on the GENESIS **table** object) is a good way of implementing linear, logarithmic, or even more exotic conversion operations where necessary.

The kinetics library also allows one to send messages to **reac** elements so as to control their rate constants. This provides a great deal of flexibility. (See Exercise 5 below.)

10.4 Summary: Molecular Computation

At the risk of stating the obvious, a few aspects of the computational properties of signaling pathways are outlined here. Parallels have been drawn between signaling systems and Boolean logic (Bray 1995), and even between neural networks (Alberts et al. 1994). Such analogies, of course, are only part of the story — otherwise, we could avoid the whole messy business of modeling chemical kinetics.

It is easy to see how a signaling pathway could perform logic operations. *And* gates could be represented as enzymes that need two or more simultaneous signals for activation. *Or* gates are enzymes that can be activated by either of two signals. *Inverters* are simply inhibitory signals. Examples of all these situations exist, but the actual biochemistry is much richer than Boolean logic.

Analog computation might be a better way to think of biochemical signaling. This would allow one to consider such signaling properties as *amplification*, as we have seen for

G proteins, *synergy*, where the effect of two signals is more than additive, *cooperativity*, where the response curve is strongly nonlinear, and so on.

Even this viewpoint has its limitations. One of the most important properties of real-world neuronal as well as biochemical signaling is time-delays. Rather than being a limitation of such signaling, this is one of the most interesting elements in the computational repertoire of such networks. *Integration* and *differentiation* of the time-course of signals now becomes possible, as does short-term *storage* of information.

The analogies could be taken to the absurd, by invoking electrical circuitry as the next level of analysis. At this point the analysis is almost as complex as the original biochemistry itself, so it doesn't help much. In any case, we still have not considered the computational implications of diffusion, compartmentalization, and the anchoring of successive enzymes on protein scaffolds. Beyond that, there are all the possibilities inherent in biochemical control of the cytoskeleton, membrane traffic, and of course the genes themselves. To put the scale of this problem into perspective, computer scientists long ago realized that self-modifying code was insanely difficult to analyze. Nature has done much better here. Imagine a computer where you could redesign the CPU, while it was running, from within software. Biochemical control of the cytoskeleton and genes is a *functioning* example of self-modifying hardware, software, and everything in between.

I regard the current state of the art in the field of biochemical signaling as similar to that in electrophysiology fifty years ago: one can begin to analyze the interactions as point processes, but limitations in the available data and the techniques currently make it difficult to construct more complex models. This is likely to change rapidly in the coming years. We are now at the point where a few specific examples of microcompartmentalization of signaling pathways have been found. These have enormous implications for the ways in which we analyze such pathways. Macroscopic rate constants and concentrations become almost irrelevant in such cases. Reaction mechanisms and pathways themselves may change in ways that are nearly impossible to duplicate in a test tube. Computational methods are one possible way in which we can begin to take test-tube data and scale them down to fit inside a dendritic spine. They are also perhaps the only way in which we can tackle the growing mountain of available data, and begin to understand the immensely complex network within the cell.

10.5 Exercises

1. Using the feedback loop model, compare results at different time steps and using explicit conservation rules. You will need to look at the *Kinetikit* on-line help manual to find out how to set up conservation rules in the model.
2. Build the G protein model from Fig. 10.2. Estimate the amplification provided by this G protein. Parameters for G proteins are readily available from the literature, or from

the library of models of signaling pathways available through the GENESIS Users Group (Appendix A.3).

3. One of the most useful operations in developing complex signaling models is to merge multiple pathways into a complex model. Groups are a very useful organizing tool in such situations. Take the feedback model you built previously, and put it all into a group. Now load in the G protein model on top of this. Can you now see a role for groups? Experiment with using the G protein model as an input pathway for the feedback model.
4. In the feedback model, one way of getting the concentration-effect curve was to provide a series of concentration steps using an **xtab** element. Try to replace the step waveform with a linearly or logarithmically increasing table. Why might you have problems here? How long would you have to run to get a reasonable answer?
5. The k_f and k_b of the **reac** object class in the kinetics library can be controlled through messages. This allows one to implement many kinds of rate equations. Implement a Hodgkin–Huxley type ion channel using this facility. This can be done both using some of the special options in *Kinetikit*, and of course directly from the script language. Estimate the execution speed of such a channel, and compare with the tabchannels, which are highly optimized.
6. Exercise on buffering (challenging!): Design a buffer system for Ca that holds its concentration at $1 \mu M$ over a wide range of added Ca. Work out how to improve the buffering by changing
 - The order of the reaction
 - Cooperativity
 - The K_d of the buffer
 - The total amount of the buffer vs. its K_d , for a given initial Ca level of $0.1 \mu M$.
 - Estimate the speed of the buffer, by seeing how well it suppresses a large but bufferable influx of Ca lasting for 10 msec .
 - Estimate how fast such a buffer would work to lower Ca levels if the buffer were suddenly released (by a flash of light) from a caged compound into a solution containing Ca.

Part II

Creating Simulations with GENESIS

Chapter 11

Constructing New Models

JAMES M. BOWER

You have now explored several tutorials intended to both provide some insights into fundamental concepts in neurobiology and to introduce you to the use of neural simulations. The remainder of this book is intended to guide you in the creation of your own simulations. In general, we have found that the most efficient way to develop new simulations is to modify one that already exists and that you understand well. We would encourage you to consider starting with a tutorial most similar to the simulation you would like to build, after having learned a few basics of GENESIS programming from the following chapters.

As discussed in Chapter 3, GENESIS has been specifically designed to allow user modification of existing simulations as well as the easy incorporation of new biological components. At the time of its development, no simulator existed that could easily support models based on the anatomical and physiological details of real nervous systems, or that could be easily modified to do so. Instead, those interested in building more realistic models of neural structures (primarily single neurons) either had to write their own special purpose code (Getting 1989, Pellionisz and Llinás 1977) or use a simulator designed for another purpose. For example, some neural modelers used electric circuit simulators originally intended for use by the electronics industry (Segev, Fleshman and Burke 1989, Segev, Fleshman, Miller and Bunow 1985). In other cases, modelers tried to adopt simulators constructed to model abstract “neural networks” of the connectionist type (Goddard, Fanti and Lynne 1987). In each case, including descriptions of real neural components was tedious, time consuming and limited.

GENESIS was designed in response to this situation. We specifically set out to build a simulator capable, in principle, of supporting any level of biological detail. Furthermore, we

wanted to ensure that as new properties and details of biological systems were discovered, they could be easily incorporated into the GENESIS system. Accordingly, this effort was fundamentally based on the assumption that the structural and physiological details of the nervous system matter. In fact, we believe that an eventual understanding of the way nervous systems compute will be very closely dependent on understanding the full details of their structure. For this reason, our own approach to modeling involves constructing computer simulations that are very closely linked to the detailed anatomical and physiological structure of the particular neuronal system being studied (Bower 1995). We refer to the resulting simulations as “structurally realistic” models to distinguish them from the more abstract types of models that have formed the principal basis for most neural modeling efforts in the past. Before beginning to describe the process of building and modifying GENESIS simulations, it will be useful to first discuss several general and specific issues relevant to our philosophy of model construction. To a considerable extent, these issues and philosophies have guided the construction of GENESIS.

11.1 Structurally Realistic Modeling

Every indication suggests that structurally realistic modeling will represent one of the largest areas of growth in computational neurobiology (Bower 1992). Continuing rapid advances in computer software and hardware technology are making it possible for neurobiologists themselves to build models. Typically, when neurobiologists are in charge of modeling efforts, they include more of the fine biological details. In addition, experimental technology has advanced to the point that the necessary information to construct realistic models can be more readily obtained.

Although technology is clearly enabling the development of more and more complex models, we do not believe that the simple capacity to make such models is the reason they should be built. Instead, as neurobiologists and modelers, we are more and more convinced that understanding, or “reverse engineering” (Bower 1995) the nervous system will critically depend on the construction of anatomically and physiologically realistic simulations.

Several practical arguments can be made in support of constructing realistic models. First, simply from the point of view of model construction and use, these simulations have several advantages over more abstract modeling efforts: for example, biological realism allows known neuroanatomical and neurophysiological data to be used as constraints for establishing model components. This is an important feature for models of complex systems that can otherwise posit an almost limitless number of components and interactions. In addition, the relationship between the parameters in a structural model and real biological measurements limits the parameter space of the models that must be explored (Bhalla and Bower 1993, Jaeger, De Schutter and Bower 1997). Because most of the time spent working with a model involves determining the dependence of its activity on model parameters, this

greatly increases the chances that modeling will yield something interesting. By generating biologically relevant outputs, modeling results are also more readily comparable to data from actual experiments, making predictions testable (Hasselmo, Barkai, Horwitz and Bergman 1994, Jaeger et al. 1997). Testability is one of the most serious problems facing all models of complex systems. Finally, the process of building such a simulation itself, in effect, highlights what information still must be obtained to build the model. This often requires the modeler to more carefully quantify exactly what is known about a particular neuron or circuit. In fact, in our experience the early stages of model building often point out more vividly what is not known about a network than what is. In this way, structural models lead naturally to ideas for additional experiments as well as provide a context in which to interpret the data once they are obtained.

Beyond the practical advantages of building realistic simulations, we believe there are also very important pedagogical reasons for constructing simulations of this kind. Specifically, we believe that, when used properly, these models have an increased chance of generating unanticipated functional insights based on emergent properties of neuronal structure. Traditionally, most models, especially those that are structurally abstract, have been less concerned with the details of neural organization than in testing a particular pre-existing theory. Those that have been concerned with neural structure still have often been primarily intent on proving biological plausibility for a particular pre-existing idea rather than using the structure of the nervous system itself to suggest new functional ideas. Although there may be nothing, in principle, wrong with these “theory demonstration” approaches, we believe that models which replicate the structure of the nervous system as a basis for exploring its computational features are more likely to uncover features that had been previously overlooked or unsuspected (cf., Hasselmo and Bower 1992, Hasselmo, Anderson and Bower 1992, Bower 1995, Bower 1997a,b). Once a new idea has emerged, abstract models can always be built to more completely explore parameter space (Hasselmo et al. 1992) and/or to cast new ideas into more traditional theoretical forms (Hasselmo 1993). However, if a realistic model comes first, it is easier to make specific physiological predictions. In our experience, it is also easier to make a realistic model abstract than it is to make an abstract model realistic. In any case, the hardest test of any model should be how well the modeler can answer both, “What do you know now that you did not know before?” as well as “How can you determine its likelihood?” On both counts, we believe biologically realistic models have a distinct advantage.

Although realistic modeling has an increasing number of converts and successes, one still often hears the criticism that the modeler in these cases is simply substituting the problem of a complex neuron or network in an animal for a complex neuron or network in a computer (cf., Churchland and Sejnowski 1988). This criticism, however, does not adequately take into account the fact that the construction of realistic models is really a process rather than an end in itself (Bower 1995, Bower 1997a,b). As mentioned, one of the principal values of this type of modeling is that it makes very clear what you do not

know and what you need to find out. Furthermore, such criticisms of realistic modeling are also often based on the assumption, or hope, that the specific details of the nervous system might not matter. As the neural theorist David Marr proposed (Marr 1982), in this view, any particular biological neuron or network should be thought of as just one implementation of a more general computational algorithm. Following a physics model, understanding the algorithm is thought to represent a more general form of understanding than can be provided by considering the details of the nervous system itself. However, it is not clear that the “technology of understanding” in the simple systems typically studied by physicists will readily apply to exceedingly complex systems like brains, which are also built as a consequence of biological evolution. For example, it is at least possible that evolutionary pressure has forced the elimination of “unnecessary components” resulting in a very tight relationship between the structure of the nervous system and its function. If correct, then a particular neural system might not represent one of many ways to perform a particular computation, but instead, could represent a much more limited solution set, or even the only solution. This is especially true if the magnitude of the computational problems solved by nervous systems is much greater than we even now realize. The dominating trend in the history of artificial intelligence is arguably the discovery of how hard these problems really are. Similarly, the recent history of engineering efforts like “neural networks” and “connectionism” has involved the development of ever more complex computing structures, despite the predisposition of those in the field toward simplicity.

For all of these reasons, we see no evidence yet to suggest that we can avoid considering all the anatomical and physiological details in our pursuit of a full understanding of nervous systems. As biologists, given the apparent complexity of these systems, coupled with our profound ignorance about how they work, or even what kind of machine they are (Nelson and Bower 1990), it seems prudent to err on the side of biology and include the details in our models, in the hope that structure will guide us to function.

11.2 The Modeling Process

Although understanding the brain may involve an eventual consideration of all its details, model building need not await a thorough description of the neural system in question. In fact, the construction of realistic simulations may be essential to the process of experimental design, aiding in the choice of which data, and therefore which experiments are currently most relevant to our evolving understanding of neural function. Neurobiology is approaching the point where the massive amount of data already obtained may more impede progress than promote it. We believe that modeling can serve as an important tool to determine which experimental investigations will yield the most useful data at a particular time. Realistic models may also represent an extremely efficient way to store the information obtained. In this regard, we are currently exploring the use of GENESIS as an actual data base for

neurobiological information. Over the next several years, we will release software to allow neurobiologists to “mine” the information about neural structures that modelers have incorporated in GENESIS.

From the point of view of the modeler, these issues come down to the question of how and where to start constructing a model. It is important to realize that this approach to modeling does not simply involve packing all known features of a particular network into a computer simulation. Even if all the information about a neuron or network were known, as it never is, including all the details at the beginning of a modeling effort is often not practical, even with today’s computers. Instead, model building is a stepwise process that we believe should start by including the minimal principal features of the network of interest. These are then added to, as necessary, to replicate specific experimental results (Bower 1995, Jaeger et al. 1997).

11.2.1 Single Neurons or Networks

From a practical point of view, the best place to start a new modeling project is with an existing model, but which model and at which level? Often the first decision that must be made is whether to begin with a detailed model of a single cell (Chapter 7) or a more general model of a network of cells (Chapters 8 or 9). As all neurons operate as parts of networks, this question comes up even in those animals with relatively small nervous systems (Selverston 1985, Harris-Warrick, Marder, Selverston and Moulins 1992).

The initial appropriate answer to this question is dependent on a number of different considerations. From the previous chapters, you should already be familiar with some of the tradeoffs in different levels of modeling. Detailed models of single cells can provide specific information about the dynamics of cellular responses to synaptic input, but do not necessarily provide any clue as to how the cell is normally synaptically activated. The more complex the cellular model, the more are the possibilities of different patterns of synaptic activation. On the other hand, models of networks usually employ more simplified neurons without the detailed channel properties that have a profound effect on cellular output and therefore on network activity. So whatever level of modeling one starts with, there are necessary abstractions leading to important limitations. For this reason, the decision about where to start usually revolves around the type of experimental data most readily available to the modeler. This is particularly true if the modeler is also an experimentalist using particular techniques, which we regard as the ideal situation.

To some extent, the question of where to start is somewhat mitigated by the likelihood that whatever level of realistic modeling one starts with, it is likely that one will eventually end up doing the other kind of modeling as well. For example, our own studies of the olfactory cortex began with network modeling (Wilson and Bower 1992), but soon required us to develop more detailed models of single pyramidal cells (Protopapas and Bower 1994). In contrast, our detailed cellular modeling of olfactory bulb mitral cells (Bhalla and Bower 1993) has

now required that we develop a network model of the bulb to more completely interpret the data. Similarly, our detailed modeling of cerebellar Purkinje cells (De Schutter and Bower 1994a,b,c, Jaeger et al. 1997) is leading us to develop a model of the cerebellar cortical circuitry to better understand the pattern of synaptic input to these neurons (Santamaría and Bower 1997). Accordingly, no matter where one starts building realistic models of a particular structure, eventually the realism of all components of the model will probably need to be enhanced. The same trend can be seen with respect to the interactions between neural structures. Thus, one motivation for originally constructed detailed simulations of the olfactory bulb's mitral cells was to provide more realistic input to the olfactory cortex model (Bhalla and Bower 1993). This single cell modeling, however, led to electrophysiological experiments (Bhalla and Bower 1997) whose interpretation now requires an increase in the realism of our olfactory cortex model to understand the effects of cortical feedback on mitral cell behavior. Although this may start to sound like a never-ending process, few biologists ever said that understanding the brain would be easy. Such statements have usually been made by engineers and physicists. Probably the most important advice we can give is to simply get started and not worry about it.

11.2.2 Modeling Steps

Retreating from the somewhat daunting prospect of eventually needing to model the entire system, there are several practical suggestions that can be made about how to start modeling, regardless of the level at which one starts. In our approach (Bower 1995, De Schutter and Bower 1994a,b), the first stage of model building usually involves establishing sufficient structural detail in the simulation to be able to replicate a set of well-characterized physiological responses. This serves to ensure, at the outset, that modeling predictions are testable experimentally. We have found that it is actually most useful to start with what we refer to as “non-physiological responses” that nevertheless reflect the overall organization of the structure in question. Responses of this type are typically experimentally generated, activating a neuron or network in some completely unnatural way. For example, in our network modeling effort with piriform cortex, we initially sought to replicate the oscillatory cortical responses obtained experimentally by artificially shocking the input pathway to the cortex (Wilson and Bower 1992). Replicating responses from these direct electrical shocks turned out to be a good initial test of the model because such massively evoked cortical activity was less dependent on the more detailed network properties that were not yet included in the simulation. In our detailed models of single cells, the initial criterion for the accuracy of a model is often its ability to replicate voltage or current clamp data (De Schutter and Bower 1994a, Jaeger et al. 1997). In this case the experimentalist has artificially injected current into a neuron with an electrode rather than with more natural synaptic activation. Although the stimulus is artificial, the responses can reflect very global features of the cell. For example, in Chapter 7 we observed a transition from bursting to single spike firing in the

Traub pyramidal cell model as the injection current was increased. Although this behavior is only seen under laboratory conditions, it is an important initial test of the model.

A second important feature of this initial stage of model building involves simulating results from as many different recording methods as possible. Thus, our piriform cortex network simulations were designed from the start to generate intracellular potentials from simulated neurons, extracellular spike train activity, and extracellular bulk electrical responses such as evoked potentials or EEGs. At the level of single cells, our Purkinje cell models not only replicate transmembrane voltage potentials, but also levels of intracellular calcium (De Schutter and Bower 1994a,b,c). The more types of responses a model generates, the more rigorously it can be tested. This effort at the beginning also ensures that the results of later simulations can be tested using diverse types of real experimental data. This is important because it is often not clear at the beginning of a modeling effort which experimental technique will provide the best means to test modeling predictions.

Once a model has been tuned on these presumably “function neutral” measures, the modeler has more confidence that the model itself includes enough of the essential features to begin exploring functional properties of the model. In practice, for single cell models, this usually involves adding synaptic conductances (De Schutter and Bower 1994b), whereas in network models it often involves introducing more complex patterns of synaptic activation or synaptic plasticity (Protopapas and Bower 1994). However, it is important to mention that even though we expected that exploring function would depend on adding these features to the models, we found that, in each of our modeling efforts, new, functionally interesting features of neural structure became apparent even in the process of tuning the model (Bower 1995, Bhalla and Bower 1993, De Schutter and Bower 1994a,b, Bower 1997a,b). This repeated experience is one basis for our claim that paying attention to the anatomical and physiological details of real neural systems is very likely to lead to important clues to their function. It seems quite likely that the relationship between the structure of nervous systems and their function may be tighter than for any other machine we know. Time and modeling will tell.

Chapter 12

Introduction to GENESIS

Programming

DAVID BEEMAN and MATTHEW A. WILSON

12.1 Simulating a Simple Compartment

Part I of this book has used several existing GENESIS simulations to introduce some of the theory underlying neural modeling. In Part II, we will build upon this background, using the GENESIS script language to create our own simulations. We will begin by simulating a simple neural compartment like that described in the first two sections of Chapter 2. Before proceeding with this tutorial, you may find it useful to review that material.

Figure 12.1 shows the equivalent electrical circuit of the basic passive neural compartment that is provided with GENESIS. Note that it is essentially the same as the “generic” neural compartment shown in Fig. 2.3, except that we have not shown the connections to neighboring compartments, nor added any variable conductance ionic channels. GENESIS provides these basic compartments, various types of channels that may be added, axonal connections to synapses and many other building blocks that are used to construct the simulation.

12.2 Getting Started with GENESIS

This chapter and the ones that follow describe the most important features and syntax of GENESIS. In order to avoid drowning you in a flood of arcane details, we will introduce the

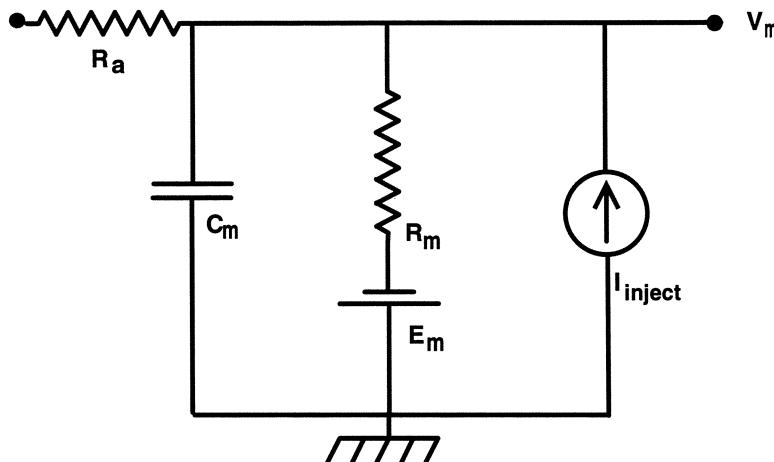


Figure 12.1 The equivalent circuit for a passive neural compartment.

use of GENESIS a little at a time, giving only the information that is needed at each stage of model building. GENESIS is continually evolving, and there will undoubtedly be new features incorporated into future versions. Although GENESIS presently runs only under the UNIX operating system with X-windows, future versions may support other operating systems and graphical interfaces.

Thus, these tutorial chapters should be used in conjunction with the GENESIS Reference Manual, which is distributed as a set of files accompanying the GENESIS distribution. These files may be used to generate the printed manual or may be viewed as part of the on-line GENESIS help facility. The manual is also provided in a hypertext version, so that it may be viewed with a web browser. The reference manual and the example scripts in the GENESIS *Scripts* directory will be periodically updated in order to keep you advised of any changes since the publication of this book. In order to be sure that you have the latest GENESIS distribution, please check the GENESIS WWW or ftp site (Appendix A).

The remainder of this chapter guides you through a brief session under GENESIS and helps you to set up and run a simple simulation. To obtain the most benefit from the following sections, you should read them while logged into a workstation on which GENESIS has been installed. At various points, you will be asked to enter GENESIS commands through the keyboard. This will let you try them out and be sure that you understand the various GENESIS commands as they are introduced.

To run the simulator, first make sure that you are at the UNIX shell command prompt. At the prompt type “genesis”. If your path is properly configured this should start up the simulator and display the opening credits. If you get a message such as “genesis: Command not found”, check your path (`echo $path`) to be sure that it contains the *genesis* directory (often `/usr/genesis`). Your home directory should contain a file named `.simrc`. If

you are still not able to run GENESIS, you or your system administrator should consult Appendix A, “Acquiring and Installing GENESIS.”

After the simulator has completed its startup procedure you should see the GENESIS command prompt, indicating that you are now in the GENESIS Script Language Interpreter (SLI). In the interpreter you can execute both UNIX shell and GENESIS commands. Try this by typing

```
ls
```

This should invoke the UNIX *ls* command, displaying files in the current directory. Typing

```
listcommands
```

should produce a list of available GENESIS commands (including *listcommands*). Note that some of these “commands,” like *cos* and *sin*, might more properly be called “functions”, as we may be more interested in the values that they return than any actions that they might perform. Nevertheless, we lump them all together as *commands*, and reserve the term *function* for a command or function that we may write ourselves in the GENESIS script language. Although there are a large number of available commands, you will typically use a much smaller subset of these.

It is also possible to combine GENESIS and UNIX shell commands. Typing

```
listcommands | more
```

will “pipe” the output of the GENESIS command *listcommands* through the UNIX command *more*, thus allowing you to page through the listing.

```
listcommands | lpr
```

will “pipe” the output to the printer and

```
listcommands > myfile
```

will redirect the output into a file called *myfile*.

12.3 GENESIS Objects and Elements

The building blocks used to create simulations under GENESIS are referred to as *elements*. Elements are created from templates called *objects* or *element types*. In order to emphasize this distinction, we give the names of GENESIS objects in boldface type and use italics for the names of the elements that are created from them. The simulator comes with a number of basic objects. To list the available objects type

```
listobjects
```

To get more information on a particular object type

```
showobject <name>
```

where “<name>” is replaced by any name from the object list.

The **compartment** object is most commonly used in GENESIS simulations to construct parts of neurons. GENESIS also has another type of compartment, the **symcompartment**, in which the axial resistance R_a shown in Fig. 12.1 is symmetrically divided between the two

sides of the compartment. However, the **compartment** is more computationally efficient to use, and is adequate for most modeling. As we will be using this object, try the command “`showobject compartment`” at this time. Most commonly used objects are documented more thoroughly with the GENESIS *help* command. For example, to obtain a detailed description of the equivalent circuit for the **compartment** object, type

```
help compartment | more
```

For more information about the GENESIS on-line documentation, simply type “`help`”.

12.3.1 Creating and Deleting Elements

To create an element from an object description, you use the *create* command. Try typing the *create* command without arguments:

```
create
```

This results in a *usage* statement that gives the proper syntax for using this command. Most commands will produce a usage statement if invoked without arguments or followed by “`--usage`”. In the case of the *create* command, the usage statement looks like

```
usage: create object name [object-specific-options]
```

In this exercise we will create a simple passive compartment. In order to keep track of the many elements that go into a simulation, each element must be given a name. To create a compartment with the name *soma*, type

```
create compartment /soma
```

Elements are maintained in a hierarchy much like that used to maintain files in the UNIX operating system. In this case, */soma* is a pathname which indicates that the soma is to be placed at the root or top of the hierarchy.

We will eventually build a fairly realistic neuron called */cell* with a soma, dendrites, channels and an axon. It would be a good idea to organize these components into a hierarchy of elements such as */cell/soma*, */cell/dend*, */cell/dend/Ex_channel*, and so on. If we do this, we need to create the appropriate type of element for */cell*. GENESIS has a **neutral** object for this sort of use. An element of this type is an empty element that performs no actions and is used chiefly as a parent element for a hierarchy of child elements.

To start the construction of our cell, give the commands

```
create neutral /cell
```

```
create compartment /cell/soma
```

As we no longer need our original element */soma*, we may delete it with the command

```
delete /soma
```

12.3.2 Examining and Modifying Elements

The commands for moving about within the GENESIS element hierarchy are similar to their UNIX counterparts. For example, to list the elements in the current level of the hierarchy

use the *le* (list elements) command

```
le
```

You should see several items listed, including the newly created *cell*.

Each element contains data fields that contain the values of parameters and state variables used by the element. To show the contents of these data fields use the *showfield* command. For example,

```
showfield /cell/soma -all
```

will display the names and contents of the data fields of the “soma,” along with some other information such as the number of incoming and outgoing messages. This example also illustrates the use of GENESIS command options. The option (“-all”) follows any command arguments and may be abbreviated to the shortest unambiguous character string, “-a”, in this case. Notice that the **compartment** object has fields *Vm*, *Em*, *Rm*, *Cm*, *Ra*, and *inject*, corresponding to the labels in Fig. 12.1 and the variables in Eq. 2.1. To display the contents of a particular field, such as the membrane resistance field *Rm*, type

```
showfield /cell/soma Rm
```

When working in GENESIS you are always located at a particular element within the hierarchy which is referred to as the *working element*. This location is used as a default for many commands that require path specifications. For example, the *le* command used above normally takes a path argument. When the path argument is omitted the working element is used and thus all elements located under the working element are listed. To move about in the hierarchy use the *ce* (change element) command. To change the current working element to the newly created soma, type

```
ce /cell/soma
```

Now you can repeat the *showfield* command used above, omitting the explicit reference to the */cell/soma* pathname:

```
showfield -all
```

This should display the contents of the soma data fields. You may find the current working element by using the *pwe* (print working element) command. Try giving the command:

```
pwe
```

Note the analogy between these commands and the UNIX commands *ls*, *cd* and *pwd*. By analogy with UNIX, GENESIS uses the symbols “.” to refer to the working element, and “..” to refer to the element above it in the hierarchy. Try using these with the *le*, *ce* and *showfield* commands. Likewise, GENESIS has *pushe* and *pope* commands to correspond to the UNIX *pushd* and *popd* commands. These provide a convenient method of changing to a new working element and returning to the previous one. Try the sequence of commands

```
pushe /cell
```

```
pwe
```

```
pope
```

```
pwe
```

The contents of the element data fields can be changed using the *setfield* command. To set the transmembrane resistance of your soma, type

```
setfield /cell/soma Rm 10
```

You can set multiple fields in a single command as in

```
setfield /cell/soma Cm 2 Em 25 inject 5
```

Now if you use a *showfield* command on the element you should see the new values appearing in the data fields.

12.4 Running a GENESIS Simulation

Most elements have the capability of performing a self-consistency test that will report problems if some aspect of the element has been improperly specified. This test is invoked with the *check* command. After the simulation is set up, it is a good idea to give the command

```
check
```

Before running a simulation the elements must be placed in a known initial state. This is done using the *reset* command which should be performed prior to all simulation runs:

```
reset
```

If you now show the value of the compartment *Vm* field with the command

```
showfield /cell/soma Vm
```

you will see that it has been reset to the value given by the parameter *Em*. To run a simulation use the *step* command, which causes the simulator to advance a given number of simulation steps. For ten steps, use

```
step 10
```

If you now display the *Vm* field, you will see that the simulator actually did something and that the value has changed from its initial value due to the current injection. This field is an example of a *state variable*. GENESIS state variables are data fields that are automatically updated by the elements when they are “run” during a simulation. Normally, these are protected with a “readonly” status so that they may be inspected, but may not be modified with the *setfield* command.

12.4.1 Adding Graphics

Although we now have a working simulation, we need a better way to output the results at each simulation step. Although GENESIS provides ways to send data to files for later analysis, most simulations will make use of some graphical output to monitor the course of a simulation. With that in mind, we will attempt to add a graph to the simulation that will display the voltage trajectory of your soma.

Graphics are implemented using graphical objects from the XODUS library that are manipulated using the same techniques described above. The *form* is the graphical object

that is used as a container for all other graphical items. XODUS forms are the “windows” that appear in the simulations which you used in the first part of this book. Thus, before making a graph, we need to make a form in which to put it. We will arbitrarily name our form `/data`, and create it from the `xform` object.

```
create xform /data
```

You may have noticed that nothing much seemed to happen. By default, forms are hidden when first created. To reveal the newly created form use the command

```
xshow /data
```

An empty window should appear somewhere on your screen. To create a graph in this form with the name *voltage*, use the command

```
create xgraph /data/voltage
```

Note that the graph was created beneath the form in the element hierarchy. This is quite important, as the hierarchy is used to define the nesting of the displayed graphical elements.

Finally, try the command

```
xhide /data
```

to hide the form and its contents. Then use the `xshow` command to bring it back again. The combination of `xshow` and `xhide` is useful for popping up and putting away menus and graphs, as we have done in the simulations in Part I of this book.

12.4.2 Linking Elements with Messages

Now you have a soma and a graph, but you need some way of passing information from one to the other. Interelement communication within GENESIS is achieved through a system of links called *messages*. Messages allow one element to access the data fields of another element. For example, to cause the graph to display the voltage of the soma you must first pass a message from the soma to the graph indicating that you would like a particular data field to be plotted. This message is established with the command

```
addmsg /cell/soma /data/voltage PLOT Vm *volts *red
```

The first two arguments give it the source and destination elements. The third argument tells it what type of message you are sending. In this case the message is a request to plot the contents of the fourth argument which is the name of the data field in the soma that you wish to be plotted. The last two arguments give the label and color to be used in plotting this field.

You can now run the simulation and view the results in the graph by giving the commands

```
reset
```

```
step 100
```

In order to plot another field in the same graph, just set up another message

```
addmsg /cell/soma /data/voltage PLOT inject *current *blue
reset
step 100
```

and you are displaying both the injection current and the voltage. If you type “step 100” again without resetting, the simulation will continue on for another 100 steps and the results will be off the scale of the graph. In the next chapter, you will learn how to set the scales for the *x-axis* and *y-axis* of the graph. However, you may always interactively change the scale for a graph axis by clicking the mouse on the axis and dragging it to higher or lower values. Hitting the “a” key while the mouse cursor is inside the graph causes the graph to automatically rescale to hold the plotted data.

Just as the *showfield* command allows you to examine the current value of data fields, the *showmsg* command lets you determine what messages have been established between elements. Both of these commands can be very useful when interactively debugging GENESIS simulations. For example, if you give the command

```
showmsg /data/voltage
```

after performing 100 simulation steps, it will produce the output

```
INCOMING MESSAGES
MSG 0 from '/cell/soma' type [0] 'PLOT' < data = 74.6631 >
< name = volts > < color = red >
MSG 1 from '/cell/soma' type [0] 'PLOT' < data = 5 >
< name = current > < color = blue >
```

OUTGOING MESSAGES

Incoming messages 0 and 1 correspond to the two messages that we sent to the graph, along with the values of the accompanying parameters after 100 steps. There are no outgoing messages from the graph. Try this command for */cell/soma* and verify that its outgoing message is consistent with the incoming message for the voltage graph.

You may remove messages with the *deletemsg* command. Try

```
deletemsg /data/voltage 1 -incoming
```

and use *showmsg* again to inspect the messages. Can you produce the same effect by deleting the corresponding outgoing message from the soma?

12.4.3 Adding Buttons to a Form

The **xbutton** graphical element is often used to invoke a function when a mouse button is clicked. Give the command

```
create xbutton /data/RESET -script reset
```

This should cause a bar labeled RESET to appear within the “data” form below the “voltage” graph. When the mouse is moved so that the cursor is within the bar and the left mouse button is clicked, the function following the argument -script is invoked. Now add another button to the form with the command

```
create xbutton /data/RUN -script "step 100"
```

In this case, the function to be executed has a parameter (the number of steps), so “step 100” must be enclosed in quotes so that the argument of -script will be treated as a single string.

At this stage, you have a complete GENESIS simulation that may be run by clicking the left mouse button on the bar labeled RESET and then on the one labeled RUN. To terminate the simulation and leave GENESIS, type either “quit” or “exit”. If you like, you may implement one of these commands with a button also.

At this time, you should use an editor to create a script containing the GENESIS commands that were used to construct this simulation. The script should begin with

```
//genesis
```

and the filename should have the extension “.g”. For example, if the script were named “tutorial1.g,” you could create the objects and set up the messages with the GENESIS command

```
tutorial1
```

If you have exited GENESIS and are back at the UNIX prompt, you may run GENESIS and bring up the simulation with the single command

```
genesis tutorial1
```

In the following chapters, we will modify this script to create a more realistic model of a neuron that may be incorporated into simple neural circuits.

12.5 How GENESIS Performs a Simulation

You should now have a complete working GENESIS simulation, assembled into a script. Let us now examine what we have created and try to understand how it works.

Notice that, unlike a program in C, Pascal, or FORTRAN, there is no explicit looping over time. Although the GENESIS script language has a *for* construct similar to that used in C, it is not used for iteration over time. When a typical GENESIS simulation script is loaded (or the commands are entered interactively) the ScriptLanguage Interpreter merely sets up the simulation when it processes these commands. It does this by creating a number of simulation elements, initializing internal data fields with the *setfield* command, and setting up messages between elements. The iteration over time is performed implicitly when the *step* command sets the simulation in motion.

When using the *showobject* command for a particular object type (a **compartment** or **xgraph**, for example), you will notice that in addition to listing the data fields of the

object and the types of messages that it may receive, it also lists the *actions* which that type of object may perform. For example, most objects are capable of performing the RESET and CHECK actions. When the *reset* or *check* command is given, each element performs the version of the RESET or CHECK action that is appropriate for its object type. Most GENESIS object types can perform the PROCESS action. This is the action that is performed 100 times when you give the *step 100* command to run the simulation. When it is performed, each element that is capable of this action goes through one iteration, checking for incoming messages and performing the computations necessary to update its internal fields.

With this *object-oriented* arrangement, each element is responsible for knowing how to perform its own actions, and affects other elements only by the exchange of messages. This modularity makes it easy to modify your simulation “on the fly,” adding and deleting compartments, ionic channels, or entire groups of cells, or changing variables to be plotted with just a few commands or the click of a mouse button.

An **xbutton** object just responds to mouse clicks, so it doesn’t perform a PROCESS action. An **xgraph** object interprets the PROCESS action by looking for incoming PLOT or PLOTSIZE messages and plotting the data that they carry as a function of the current simulation time. For a **compartment** object, “PROCESS” means to look for certain types of incoming messages and to perform the calculations needed to update the *Vm* field. This is done using the current values of its own data fields and any data accompanying incoming messages.

In other chapters, you will be introduced to some of the types of messages that various objects may receive. In general, the name of the message tells the receiving element what to do with the data that are sent with the message. Sometimes a message is used simply to set an internal data field of an element. For example, the INJECT message may be used as a way to set the *inject* field of a compartment at each simulation step. This would be used instead of the *setfield* command if we wanted to continuously vary the current injection during the course of a simulation. For example, a **pulsegen** object (a pulse generator) could send its output state with an INJECT message to the soma, in order to provide pulses of current injection, rather than the steady injection current that we used in this simulation. You can find examples of this use of the **pulsegen** object in the *inputs.g* files that form part of the *Cable* and *Neuron* tutorials.

Further details of the internal operation of GENESIS are given in the GENESIS Reference Manual chapter “Customizing GENESIS,” which describes how to create your own objects.

12.6 Exercises

1. Add a button that will allow you to exit the simulation.

2. Try some experiments with sending messages to `/cell/soma`, instead of setting the `Em` or `inject` fields. You will need some element to send the message, and a field value to be sent. An easy way to accomplish this is to use the `x`, `y` or `z` field value of the `/cell` element. Although these fields are normally used for positioning the cell in a network, they may be used for any purpose. Try setting the `x` field of `/cell` to 5 and the `inject` field of `/cell/soma` to zero. By using an INJECT message from `/cell` to `/cell/soma`, can you produce results that are equivalent to setting the `inject` field to 5? Does the message affect the actual value of this field? What does the EREST message do?

Chapter 13

Simulating a Neuron Soma

DAVID BEEMAN

13.1 Some GENESIS Script Language Conventions

In the previous chapter, we modeled the simple passive compartment shown in Fig. 12.1. The membrane resistance R_m is in series with a leakage battery E_{rest} , and is in parallel with a membrane capacitance C_m and a current source I_{inject} . Assembled into a script, the commands used would look something like the listing shown in Fig. 13.1. Note the use of “//” for comments. Multi-line comments may be entered by bracketing the commented lines with “/*” and “*/”, as in C. This script also illustrates the use of the backslash character, “\\”, in order to continue a statement (the second *addmsg* command) to another line. This will be useful when entering lengthy *setfield* commands.

In this chapter, we choose cell parameters that are typical of a neuron soma. We will then add voltage-dependent sodium and potassium ion channels to the compartment. Future chapters extend this to a complete multi-compartmental model of a neuron. The necessary background for understanding these sorts of models may be found in Part I of this book.

13.1.1 Defining Functions in GENESIS

As with any programming language, the GENESIS script language allows you to define functions in order to make your programs more modular and easier to modify. These functions should be grouped together at the beginning of the script, preceding any statements that use them. Because we will begin by modifying the simple compartment used in the previous chapter to make it look more like a neuron soma, it is a good idea to rewrite the

```

//genesis - tutorial1.g
// create a parent element
create neutral /cell

// create an instance of the compartment object
create compartment /cell/soma

// set some internal fields
setfield /cell/soma Rm 10 Cm 2 Em 25 inject 5

// create and display a graph inside a form
create xform /data
create xgraph /data/voltage
xshow /data

// send a message (PLOT Vm) to the graph
addmsg /cell/soma /data/voltage PLOT Vm *volts *red
addmsg /cell/soma /data/voltage \
    PLOT inject *current *blue

// make some buttons to execute simulation commands
create xbutton /data/RESET -script reset
create xbutton /data/RUN -script "step 100"
create xbutton /data/QUIT -script quit

check      // perform a consistency check for each element
reset     // initialize each element before starting the simulation

```

Figure 13.1 A GENESIS script to simulate a simple compartment with current injection.

script to use functions for the creation of the soma and the graph. The general form of a function in GENESIS is

```

function <function_name>(arg1, arg2, ...)
    type1 arg1
    type2 arg2
    .
    .
    type_lv1 local_var1
    .
    <commands>
    .
end

```

Here, the data types are one of *int*, *float* or *str*. As in C, arguments are passed by value. An example of a GENESIS function would be:

```
function print_area(length, diameter)
    float length, diameter
    float area
    float PI=3.14159
    area = PI*diameter*length
    echo The area is {area}
end
```

The command

```
print_area 5 5
```

should produce an output something like

```
The area is 78.5397
```

If you would like the result formatted to show fewer significant figures, try generating the usage statement for the *floatformat* command, or use the *help* command to get further information.

Note that when a GENESIS function is invoked, its arguments are given as a list, separated by spaces (the *command line form*), rather than as a list in parentheses separated by commas (the *function call form*).

This example also illustrates how one sets the value of a variable (*PI*) at the time that it is declared, as well as the use of braces (“curly brackets”). The predefined GENESIS function *echo* prints out its list of arguments to the screen. The braces around “area” cause it to be evaluated to the value represented by the variable *area*, rather than the string of characters “area”. The following statements both produce the same output:

```
echo The area is {area}
echo "The area is "{area}
```

In the first statement, the function *echo* has four arguments. In the second, it has only one argument. The only time you may omit the braces when a variable name is to be evaluated is when it is used in an arithmetic expression. Thus, the following two statements are equivalent:

```
area = PI*diameter*length
area = {PI*diameter*length}
```

This is a point of much confusion among beginning GENESIS programmers, so it would be a good idea to experiment with writing some simple functions that evaluate and echo the values of variables. Some further examples are given in the GENESIS Reference Manual.

Once you feel comfortable writing GENESIS functions, copy your script to a new file, *tutorial2.g*, and modify it so that a function is used to create the soma compartment. For maximum generality, define a function *makecompartment* that takes an element path name as an argument. Then, the command “`makecompartment /cell/soma`” should create the soma compartment beneath the parent element */cell* and set the internal fields. Also write a function *make_Vmgraph* to create and show the graph for *Vm* with its associated form and buttons. Once you have completed these changes to your script, verify that the simulation still works as before.

13.2 Making a More Realistic Soma Compartment

13.2.1 Some Remarks on Units

The internal fields used in GENESIS elements have no implicit units. In the statement “`setfield /cell/soma Rm 10 Cm 2 Em 25 inject 5`”, the *Rm*, *Cm* and *Em* fields could be in ohms, farads and volts. On the other hand, their values (10, 2 and 25) are of magnitudes that might more appropriately be expressed in kilohms ($K\Omega$), microfarads (μF) and millivolts (mV). The choices for these units will determine the units of time and current. Any inconsistency in the units that are used can result in confusion as well as incorrect results! One way to keep confusion to a minimum is to stick to SI (MKS) units. This is the approach taken with the *Neurokit* program and its associated prototype libraries of cell components. Unfortunately, quantities typical of cells tend to have either very large or very small values when expressed in SI units. For this reason, many people prefer to use *physiological units*. This approach was taken in the *MultiCell*, *Neuron* and *Squid* simulations. Table 13.1 may help you to keep the units straight. In this book, we stick to SI units.

Once having settled upon a consistent set of units, we need to find a way to determine the cell parameters, membrane resistance (R_m), membrane capacitance (C_m) and axial resistance (R_a) for a compartment of given dimensions. In order to specify parameters that are independent of the cell dimensions, *specific units* are used. For a cylindrical compartment, the membrane resistance is inversely proportional to the area of the cylinder, so we define a *specific membrane resistance* R_M , which has units of $\Omega \cdot m^2$. The membrane capacitance is proportional to the area, so it is expressed in terms of a *specific membrane capacitance* C_M , with units of farads per square meter. In future chapters, compartments are connected to each other through their axial resistances R_a . The axial resistance of a cylindrical compartment is proportional to its length and inversely proportional to its cross-sectional area. Therefore, we define the *specific axial resistance* R_A to have units of $\Omega \cdot m$.

<i>Quantity</i>	<i>SI units</i>	<i>physiological units</i>
resistance	ohm (Ω)	kilohm ($K\Omega = 10^3 \Omega$)
capacitance	farad (F)	microfarad ($\mu F = 10^{-6} F$)
voltage	volt (V)	millivolt ($mV = 10^{-3} V$)
current	ampere (A)	microampere ($\mu A = 10^{-6} A$)
time	second (sec)	millisecond ($msec = 10^{-3} sec$)
conductance	siemen ($S = 1/\Omega$)	millisiemen ($mS = 10^{-3} S$)
length	meter (m)	centimeter ($cm = 10^{-2} m$)

Table 13.1 Correspondence between SI and physiological units for common quantities used in neural modeling.

For a compartment of length l and diameter d we then have

$$R_m = \frac{R_M}{\pi l d} \quad (13.1)$$

$$C_m = \pi l d C_M \quad (13.2)$$

$$R_a = \frac{4l R_A}{\pi d^2}. \quad (13.3)$$

13.2.2 Building a “Squid-Like” Soma

Our goal is to build a cylindrical soma compartment that has the same physiological properties as those of the squid giant axon studied by Hodgkin and Huxley (1952d). We will make our soma smaller, with both the length and diameter equal to $30 \mu m$, but will use the same specific resistances and capacitances. (Note that when the length and diameter are the same, this will have the same surface area as a spherical soma.)

Therefore, we will begin our modification of the script by declaring and setting some global variables at the very beginning, before the function definitions:

```
// soma parameters - chosen to be the same as in SQUID (but in SI units)
float RM = 0.33333           // specific membrane resistance (ohms m^2)
float CM = 0.01               // specific membrane capacitance (farads/m^2)
float RA = 0.3                 // specific axial resistance (ohms m)

// cell dimensions (meters)
float soma_l = 30e-6          // cylinder equivalent to 30 micron sphere
float soma_d = 30e-6
```

We also need to set some potentials. Considering the outside of the cell to be at zero potential, the resting potential inside the soma should be at $-70 mV$. For consistency

with the notation used in many GENESIS scripts, we call this variable *EREST_ACT*. In many simulations, this would be the value of E_m , the “battery” in series with the membrane resistance. However, Hodgkin and Huxley found it necessary to set E_m to a leakage potential E_{leak} that compensates for current flow through other channels (such as chloride channels) which were not explicitly taken into account in their model. E_{leak} is set to a value that results in no net current flow when the cell is at *EREST_ACT*. This results in E_{leak} being 10.6 mV more positive than *EREST_ACT*. Although we will add the ion channels later, now is a good time to define and set the sodium and potassium equilibrium potentials, which we will call *ENA* and *EK*. The script should now contain the additional statements

```
float EREST_ACT = -0.07           // resting membrane potential (volts)
float Eleak = EREST_ACT + 0.0106 // membrane leakage potential (volts)
float ENA    = 0.045             // sodium equilibrium potential
float EK     = -0.082            // potassium equilibrium potential
```

Once these variables have been declared and initialized, modify your *makecompartment* function to take additional arguments for the compartment length, diameter and rest potential. It should then set the *Em* field to the rest potential and calculate and set *Rm*, *Cm* and *Ra* using *RM*, *CM*, *RA* and the compartment dimensions. (As we are not connecting this compartment to another through its axial resistance, the value of *Ra* is irrelevant. Nevertheless, we might as well give it the correct value.) As we would like to make this function general enough to use for creating a dendrite compartment later, the *inject* field should be set after the function is invoked, rather than being set within the function definition. An appropriate value of the injection current for this simulation is 0.3 nA, so we should be able to create a soma with fields set to the proper values using the statements

```
create neutral /cell
makecompartment /cell/soma {soma_1} {soma_d} {Eleak}
setfield /cell/soma inject 0.3e-9
```

Before we can plot the results of the simulation, we need to be sure that the graph */data/voltage* is properly scaled. The deceptive simplicity of the previous simulation stems from the fact that the parameters were chosen so that the voltages and times fell within ranges that were consistent with the default values of the fields *xmin*, *xmax*, *ymin* and *ymax* for the **xgraph** object. These values may be inspected with the command

```
showfield /data/voltage -a
```

or

```
showfield /data/voltage xmin xmax ymin ymax
```

Likewise, the *setfield* command can be used to set these fields. When we later add voltage-activated ion channels to our model, we will expect to see action potentials that extend from slightly below the resting potential to a slightly positive value. A reasonable time scale to observe a few action potentials would be about 100 msec. In order to make it easy to modify these ranges, we can define some local variables in the *make_Vmgraph* function and modify the relevant part of the script to look something like this:

```
float vmin = -0.100
float vmax = 0.05
float tmax = 0.100 // default simulation time
create xform /data
create xgraph /data/voltage
setfield ^ xmax {tmax} ymin {vmin} ymax {vmax}
```

The caret symbol (“^”) is a convenient shorthand to refer to the most recently created element. We could also have used

```
setfield /data/voltage xmax {tmax} ymin {vmin} ymax {vmax}
```

As the vertical scale is no longer appropriate for the plotting of the injection current, you should delete the line that sets up the message to plot the *inject* field. Once these changes have been entered and you have successfully loaded the script with no reports of syntax errors from GENESIS, click the left mouse button on the RUN button. Were the results what you expected? They probably were not.

13.2.3 GIGO (Garbage In, Garbage Out)

It is now time for a step in the construction of this simulation that has been delayed for far too long. Before performing any sort of computer simulation, you should analyze the situation and try to predict the main features of the results. Afterwards, look at the simulation results with a critical eye in order to resolve any differences between what you see and what you expected. If the results of the simulation are not what you expect, it is time for more thought. Either your understanding of the processes occurring in the system is incorrect (or incomplete), or there is something wrong with the program. In the former case, these sorts of surprises provide one of the main motivations for performing “computer experiments.” By finding explanations for these unexpected results, we have used the simulation to increase our understanding of the system. In this case, the flat horizontal line seen in the V_m plot is an indication that we have neglected something important.

The GENESIS command “`help compartment | more`” will remind you of the equivalent circuit that we are modeling and the differential equation that is being solved. The on-line help shows a circuit diagram and an equation that are equivalent to Fig. 2.3 and Eq. 2.1.

The diagram reveals that the current I_{inject} flows through R_m to create a potential difference that is in series with E_m . (It also shows a variable channel conductance G_k and its associated equilibrium potential E_k but we have not added the ion channels yet.) Without the ion channels or adjacent compartments, it should be a simple matter for you to calculate the steady-state value of V_m . In this case, the equivalent circuit is reduced to that shown in Fig. 12.1, and Eq. 2.1 becomes

$$C_m \frac{dV_m}{dt} = \frac{(E_m - V_m)}{R_m} + I_{inject}. \quad (13.4)$$

Initially, V_m will equal E_m , and the steady state will be reached after a time given roughly by the time constant for charging the membrane capacitance, $\tau = R_m C_m$. You should now give the command “`showfield /cell/soma -a`” in order to inspect the values of these quantities and make some rough calculations. (HINT: You should conclude that V_m will level off at about -0.024 V, with a time constant of about 0.0033 sec. If your results are significantly different, you should check the way in which you calculated R_m and C_m from R_M and C_M .) The problem with our simulation seems to lie in the time dependence. Rather than asymptotically reaching the final value of V_m over the 100 simulation steps, we reach it after the first step.

Perhaps you have anticipated this result. The simulator has performed a stepwise numerical integration of a differential equation over 100 time steps. However, there has been no mention of the time interval used for each integration step. The section in the GENESIS Reference Manual on clocks discusses the commands `getclock`, `setclock` and `useclock`. One may specify up to 100 different clocks that have their time steps set by a command of the form “`setclock <clock># <stepsize>`”. For example,

```
setclock 0 0.001
```

Clock number 0 is the global simulation clock that we need to set. The default step size is 1.0 in whatever units we are using. This was fine for the time scale that we used in the previous chapter, but is clearly much too large for the present simulation. When designing simulations, you will need to give some thought to the issue of picking an appropriate simulation step size. To some degree, this will be determined by experiment. As a starting point, you should pick a step size that would allow you to draw a smooth curve if you were to make a “connect-the-dots” type plot of the most rapidly varying variable at each time step. If the step size is adequately small, decreasing the size should produce no change in the results. Of course, using too small an integration step can needlessly slow down your simulation and become a source of round off error. As a practical consideration, you might have to decide what is a “tolerable” amount of difference from the ideal.

You can experiment with different step sizes by interactively issuing the `setclock` command to the GENESIS prompt. If you vary the step size, it will be more convenient for you to specify an amount of time for which the simulation should run, rather than a number of

steps to be performed. Fortunately, this may be accomplished by using the `-time` option of the `step` command. Modify the statement that creates the RUN button to read

```
create xbutton /data/RUN -script "step \"{tmax}\" -time"
```

Notice the use of spaces within the quotes. This prevents the command `step` from running into the value of the variable `tmax`. Likewise, the space before the option string “`-time`” separates it from the time value. If you have any doubts as to whether the string is being parsed correctly, use the `showfield` command to examine the `script` field of the button. In this case, it should evaluate to “`step 0.1 -time`”. Often it is easiest to avoid the complexities of building up a command string in this manner by defining a special-purpose function to do the job. For example:

```
str tmax = 0.1 // define a global variable for the run time
.
.
function step_tmax
    step {tmax} -t
end
.
.
function make_Vmgraph
.
    create xbutton /data/RUN -script step_tmax
.
end // make_Vmgraph
```

Although indiscriminate use of global variables in a program or simulation script should be avoided, the use of the `step_tmax` function with a global variable `tmax` lets you easily change the run time for the simulation. Can you think of an easy way to change the time scale of the graph whenever you change `tmax`?

Once you have found a satisfactory step size, set the simulation clock to this value within your script. You should now have a properly running (but boring) simulation of a passive soma compartment with no voltage activated channels. In the next chapter, we will add some ion channels in order to create action potentials. However, we will first offer some suggestions for tracking down errors in GENESIS simulations.

13.3 Debugging GENESIS Scripts

The GENESIS SLI provides descriptive error messages for most errors in syntax. For example, a misspelling in the line in Fig. 13.1 that creates the soma compartment might cause it to read

```
create compartment /cell/soma
```

You would then receive the messages

```
<tutorial1> line 7
** Error - could not find object 'compartement'
unable to create 'soma'
<tutorial1> line 18
** Error - addmsg : cannot find element '/cell/soma'
<tutorial1> line 19
** Error - addmsg : cannot find element '/cell/soma'
```

Notice that two additional errors occurred because the first one prevented */cell/soma* from being created. At some point your simulation scripts and their bugs will become complex enough that the interpreter will not recognize the exact error and will abort the simulation with only a message to the effect that a syntax error has occurred. In other cases, the scripts will be syntactically correct and will execute, but will produce obviously incorrect results.

The interactive nature of the SLI makes it easy to track down the point at which the simulation went astray. For example, the *le* command will let you know how far you got in creating the simulation elements before a fatal error was encountered. The *showmsg* command will let you see whether the desired linkages were set up between these elements. The *listglobals* command will list the names and values of any variables or functions that had been declared. If everything seems to be in place and properly connected, then use *step* to single step through the simulation, and use *showfield* to see if the element fields are being set to the proper values. As with any programming language, you may embed temporary print (*echo*) commands at critical points in the program to print out status information.

GENESIS also has a *debug* command that takes an integer argument to set the *debug level*. When used with no arguments, it displays the current debug level. The default is level 0. For level 1 or higher, most objects produce additional status information. Typically, increasing the number will increase the amount of information displayed. Unfortunately, a simulation that runs for more than a few steps may flood you with more output than you want. Thus, it is best to perform a single step at a time when using a non-zero debug level.

13.4 Exercises

1. In the *Neuron* tutorial (Chapter 6) the compartment-specific parameters R_M , R_A , C_M and compartment dimensions are given in physiological units. Using the tutorial, inspect the parameters and dimensions of the soma compartment and calculate the values of R_m , R_a and C_m in $K\Omega$ and μF .
2. Calculate the steady-state value of V_m that is expected in this simulation.

Chapter 14

Adding Voltage-Activated Channels

DAVID BEEMAN

14.1 Review

In the previous tutorial, we created a neuron soma compartment having the same physiological properties as those of the squid giant axon studied by Hodgkin and Huxley (1952d). In this tutorial, we will add voltage activated sodium and potassium channels to the soma by modifying a copy of the script that you produced in the previous tutorial. Before continuing with this tutorial, you may wish to review the discussion of the Hodgkin–Huxley channel models in Chapter 4. In other tutorials, we will build upon this to produce multi-compartmental models of neurons and networks of these neurons. Your script from the previous tutorial should look something like the listing of *tutorial2.g* in Appendix B.

A number of conventions have been used in this example script in order to illustrate some principles of good GENESIS programming style. Any declarations and assignments of global variables are performed at the beginning of the script, followed by function definitions, and then the main body of the script. Although functions to create XODUS forms and their associated widgets often tend to be rather specific, it is best to make generally applicable functions whenever possible, as we have done with the *makecompartment* function. If a variable is not likely to be used outside a particular function, it should be declared locally to the function. Of course, the liberal use of comments will make it easier for you or others to understand your scripts. Your own script is unlikely to be exactly like this example. However, if it has diverged too much from this general style, now might be a good time for a rewrite and “cleanup.” During the course of development of a simulation, hindsight can

be valuable. It is best to frequently go back and rewrite early code, rather than continuing to build upon programs that could have been written more cleanly.

14.2 More Fun With XODUS

In previous chapters, we have learned how to use three XODUS graphical objects, or *widgets* that are available within GENESIS. These objects (**xform**, **xgraph** and **xbutton**) and others are well documented in the GENESIS Reference Manual. As you develop more sophisticated simulations and need to exercise more control over the placement, size and labeling of graphical elements, you will want to read about the many options that can be set when these elements are created.

For example, the default title appearing on the graph (graph) may be changed by using the **-title** option when the graph is created. The labels on the buttons were set by default to the name of the **xbutton** element. Usually you can choose an appropriate name to correspond to the label you want, but you may gain additional flexibility by setting the *onlabel* and *offlabel* fields. Typing “**showobject xbutton | more**” will give you a list of the fields that you can set to modify the appearance of the buttons. You may also wish to change the size and location of the form and the elements contained within it. For example, the **RESET** and **RUN** buttons could be placed side by side. It would also be a good idea to isolate the control buttons to a “control” form like those appearing in many of the simulations we used in Part I of the book. If we remove the control buttons from the *Vmgraph* form, the **make_Vmgraph** function will be more versatile and can be used in other simulations that we create.

The following statements illustrate some ways in which we could change the layout of a form called */control* that contains the control buttons we have used so far.

```
function make_control
    create xform /control [10,50,250,145]
    create xlabel /control/label -hgeom 50 -bg cyan -label "CONTROL PANEL"
    create xbutton /control/RESET -wgeom 33%           -script reset
    create xbutton /control/RUN   -xgeom 0:RESET -ygeom 0:label -wgeom 33% \
        -script step_tmax
    create xbutton /control/QUIT -xgeom 0:RUN -ygeom 0:label -wgeom 34% \
        -script quit
    xshow /control
end
```

The horizontal and vertical positions and the width and height of an XODUS widget are specified by the four *geometry* fields *xgeom*, *ygeom*, *wgeom* and *hgeom*. These fields may be set at any time with the *setfield* command. You may also set them when the widget is created, by using the options **-xgeom**, **-ygeom**, **-wgeom** and **-hgeom**. The first line

illustrates a shorthand notation for specifying these four options. The four numbers in the square brackets represent the four geometry fields for the form, measured in pixels. For **xform** elements, positions are measured relative to the upper left-hand corner of the screen. In this case, we've given the form some extra height to save room for widgets that we might want to add later. You may want to experiment a bit to find the best size for your forms. One way to do this is to use the mouse to resize a form that you have created (usually by clicking and dragging the mouse on the icon at the right end of the form's title bar). Then use *showfield* to find the resulting geometry.

For other widgets, positions are measured relative to the upper left-hand corner of the form that contains them. If these aren't specified, they are set by default to be at the left-hand side of the parent form and just below the previously created widget. The default width *wgeom* is the width of the parent form. Each type of widget has its own default height *hgeom*.

The second line creates another type of widget, the **xlabel**. It is simply a rectangular box containing a string of text. It performs no action when you click on it. As only the *hgeom* field was specified when the label was created, it will have the width of the form and will be just below the top of the form. It will be 50 pixels high and will have the label "CONTROL PANEL". The option "-bg cyan" sets the background of the label to a color that will clash nicely with the default colors of the other widgets. You might want to experiment with other color schemes! You may do this interactively for the various widgets by using *setfield* to set the *bg* field.

A percent sign may be used after a number to indicate that it represents a percentage of the screen or form dimension. This is used to create a RESET button that is 33% of the width of the form. The statement that creates the RUN button illustrates another way to specify a position. The notation "-xgeom 0:RESET" indicates that the button is to be placed 0 pixels to the right of the RESET button. The option "-ygeom 0:label" means that it will be placed just below the label. What would happen if this specification were omitted? Note that as we suggested in the previous chapter, we have used a *step_tmax* function to run the simulation for a time *tmax*.

After removing the lines in *make_Vmgraph* that create these buttons and adding some extra "bells and whistles," your function might look like

```
function make_Vmgraph
    float vmin = -0.100
    float vmax = 0.05
    create xform /data [260,50,350,350]
    create xlabel /data/label -hgeom 10% \
        -label "Soma with Na and K Channels"
    create xgraph /data/voltage -hgeom 90% -title "Membrane Potential"
    setfield ^ XUnits sec YUnits Volts
    setfield ^ xmax {tmax} ymin {vmin} ymax {vmax}
```

```
xshow /data
end
```

In this example, the default title has been replaced with a more descriptive one. In addition, the internal fields of the **xgraph** element, *XUnits* and *YUnits*, have been set to provide meaningful horizontal and vertical axis labels. Try modifying your simulation script to use these new *make_control* and *make_Vmgraph* functions and observe the effect that they have.

The GENESIS Reference Manual describes some other widget fields that you may set to specify colors, font sizes and alternate labels. Further examples of the use of XODUS widgets may be found in the various *forms.g* scripts used in the tutorial simulations from Part I of this book. The modularity of GENESIS makes it easy to “steal” useful functions or bits of code from these simulations. For example, the *Cable*, *Neuron*, and *MultiCell* simulations all contain a script *xtools.g*. This contains a useful function (*makegraphscale*) to pop up a menu for changing the scale of the graphs that are created in their *forms.g* scripts. The *Squid* simulation uses a slightly different function to accomplish the same thing.

If you prefer, you may leave these details of “prettifying your display” until after you have learned to add channels to the model. However, we should now spend a few minutes learning to use another useful XODUS object, the *dialog widget*. The following statement creates an element called *Injection* within the */control* form, using the **xdialog** object.

```
create xdialog /control/Injection -label "Injection (amperes)" \
              -value 0.3e-9 -script "set_inject <widget>"
```

This element consists of a label adjacent to a dialog box in which the user can edit or enter text. The *-label* option is followed by the string used for the label in place of the default, which would be the name of the element. The backslash “\” indicates a continuation of the *create* statement to another line. The argument of the *-value* option is the initial value that appears in the dialog box and is stored in the *value* field of the element. As with any other element, this field may be inspected and set with the *showfield* and *setfield* commands. In addition, the value may be edited within the dialog box. When the user hits “Return” while the cursor is within the dialog box, the *value* field is set. In addition, this causes the function used as the argument to *-script* to be executed. Here, *set_inject* is a function that we will define. (NOTE: it is a common mistake to change the contents of the dialog box without hitting “Return”. In this case, the old value is still being used.)

The argument of *-script* in our dialog box example employs another useful bit of shorthand. The notation *<widget>* stands for the widget that is being defined. Thus,

```
-script "set_inject <widget>"
```

is equivalent to

```
-script "set_inject /control/Injection"
```

In our example, we would like to define a function that gets the variable stored in the *value* field of the **xdialog** element and uses it to set the *inject* field of our soma compartment. The following function takes the name of the **xdialog** element as its argument and uses the GENESIS *getfield* function to return the value of the *value* field.

```
function set_inject(dialog)
    str dialog
    setfield /cell/soma inject {getfield {dialog} value}
end
```

The *getfield* function is another useful command for dealing with the internal data fields of an element. It takes the name of the element and the name of the field as arguments and returns the value of the field. Thus, the statement

```
echo {getfield /control/Injection value}
```

would print out the value of the dialog box. Likewise,

```
set_inject /control/Injection
```

will use *getfield* to retrieve this value and *setfield* to set the *inject* field of */cell/soma*. If you make these changes to your script, you will now be able to change the soma injection current by using the dialog box.

14.3 Voltage-Activated Channel Objects

GENESIS provides several different types of objects for implementing voltage dependent ion channels. The **hh_channel** object, which we will use in this tutorial, provides the simplest way to implement the equations used by Hodgkin and Huxley to model sodium and potassium channels in the squid giant axon. Another object, the **vdep_channel**, may be linked to **vdep_gate** objects to create channels that have a more general form of the equations. This latter approach is somewhat slower because messages must be passed between the separate channel and gate objects. For maximum generality in channel modeling, one may use the **tabchannel** or **tab2Dchannel** object, or a combination of **tabgate** objects used with **vdep_channel** objects. The **tabchannel**, **tab2Dchannel** and **tabgate**, which are described in Chapter 19, allow the use of tables with interpolation rather than equations to represent the voltage dependencies. This makes it possible to model channels using experimental data that do not fit the Hodgkin–Huxley form. For serious modeling, we strongly recommend the use of **tabchannel** or **tab2Dchannel** objects. As we will see in Chapters 19 and 20, this is particularly important for accurate simulation of cells containing a large number of compartments. The GENESIS directory *Scripts/neurokit/prototypes* contains scripts that illustrate the use of all of these ways to implement voltage-dependent channels.

14.3.1 The **hh_channel** Object

In the Hodgkin–Huxley model, the general form for the channel conductance is represented as being proportional to an activation state variable raised to an integer power, times an inactivation state variable raised to another integer power. The **hh_channel** object calculates the channel conductance from the equation

$$Gk = Gbar \cdot X^{Xpower} Y^{Ypower}. \quad (14.1)$$

The **hh_channel** fields in Eq. 14.1 correspond to the variables in Eqs. 4.8 and 4.9. In the usual Hodgkin–Huxley notation for the Na channel, the activation state variable m corresponds to our variable X , with $Xpower = 3$, and the inactivation variable h corresponds to Y with $Ypower = 1$. The Hodgkin–Huxley K channel activation variable n is represented by X in our notation, with $Xpower = 4$. The K channel has no inactivation state variable, so we use $Ypower = 0$. As we are using SI units, the constant of proportionality $Gbar$ (\bar{g}) should be expressed in siemens (1/ohms).

GENESIS uses the convention that a positive current represents a flow of positive charge into the compartment, so the current through the channel is given by

$$Ik = Gk(Ek - Vm). \quad (14.2)$$

Here, Vm is the membrane potential of the compartment that contains the channel, and the other variables used in these equations are the names of fields of the **hh_channel** object. These correspond to the subscripted variables in Eq. 4.2. The field Ek represents the value of the ionic equilibrium potential, which we will express in volts. Channel elements that are created from the **hh_channel** object calculate both X and Y by solving differential equations of the form

$$\frac{dX}{dt} = \alpha(1 - X) - \beta X, \quad (14.3)$$

corresponding to Eqs. 4.11–4.13. The voltage-dependent rate constants α and β can each assume one of the three functional forms:

FORM 1 (EXPONENTIAL):

$$\alpha(V_m) = A \exp\left(\frac{V_m - V_0}{B}\right) \quad (14.4)$$

FORM 2 (SIGMOID):

$$\alpha(V_m) = A / \left(\exp\left(\frac{V_m - V_0}{B}\right) + 1 \right) \quad (14.5)$$

FORM 3 (LINOID):

$$\alpha(V_m) = A(V_m - V_0) / \left(\exp\left(\frac{V_m - V_0}{B}\right) - 1 \right). \quad (14.6)$$

Note that Eqs. 4.23, 4.24 and 4.26-4.29, used by Hodgkin and Huxley to fit the K and Na channel rate constants, each fall into one of these three forms. The form to be used and the constants A , B , and V_0 are specified for each rate constant by setting the corresponding fields in the **hh_channel** element. These are:

- (α for X): X_alpha_FORM , X_alpha_A , X_alpha_B , X_alpha_V0
- (β for X): X_beta_FORM , X_beta_A , X_beta_B , X_beta_V0
- (α for Y): Y_alpha_FORM , Y_alpha_A , Y_alpha_B , Y_alpha_V0
- (β for Y): Y_beta_FORM , Y_beta_A , Y_beta_B , Y_beta_V0 .

14.3.2 Adding Hodgkin–Huxley Na and K Channels to the Soma

In order to simplify the process of setting all these internal fields, we will borrow a script from the *Neurokit* library of channel prototypes. The file *hhchan.g*, which is listed in Appendix B, defines two functions, *make_Na_squid_hh* and *make_K_squid_hh*. These create the two channels from **hh_channel** objects and set the internal fields to the proper values. In order to understand how we will use these functions, you should now take a look at the *hhchan.g* script.

The fields X_alpha_FORM , X_beta_FORM , etc. take integer values (1, 2 or 3) to specify which form of the rate constant equations to use. For clarity, it is more desirable to refer to these forms by name, as is done in *hhchan.g*. In order to allow this, *hhchan.g* contains the statements

```
int EXPONENTIAL = 1
int SIGMOID      = 2
int LINOID       = 3
```

at the beginning of the script. The contents of the *hhchan.g* script can be made part of our simulation by using the GENESIS *include* command “*include hhchan*” after these definitions. This script is found in the GENESIS *Scripts/neurokit/prototypes* directory. Normally, the *.simrc* file in your home directory will set the GENESIS environment variable “*SIMPATH*” so that the *prototypes* directory will be searched when a file is specified for inclusion. Thus, you may use this or one of the other channel “prototype” scripts without having a copy in the directory that contains your simulation scripts.

As with any modern computer language, you can make your GENESIS simulations much easier to understand by breaking them into modules that are combined by the use of *include*. For example, large GENESIS simulations often put the graphics functions together in a script called *forms.g*, so that it may be included if graphics are to be used, and may be omitted if the simulation is to be performed in the background with output to files. Although GENESIS makes no distinction between constants and variables, it is often useful to include

a file with a name like *constants.g* at the beginning of the main simulation script. This will contain assignments of global variables, such as ionic equilibrium potentials, which are not expected to change during the course of a simulation. The *MultiCell* demonstration in the *Scripts* directory illustrates this structure.

As our simulation script will be relatively short, we will only use the one included script *hhchan.g*, but will try to keep our global variables and constants together at the beginning of the script. As we add more graphics-related function definitions, it will be useful to keep them together in one part of the script, rather than interspersing them with non-graphics functions. However, looking at the listing for *hhchan.g*, we see that it makes a necessary exception to the guideline of keeping global variables together in one place.

After some initial comments, the script defines and initializes some variables (*ER-EST_ACT*, *ENA*, *EK*, and *SOMA_A*) for the membrane resting membrane potential, channel equilibrium potentials, and the area of the soma. These are needed by the following function definitions. As this script and the other channel prototype scripts were intended to be available for inclusion in many different simulations, it makes sense to define the needed “constants” in these scripts. In the case of *hhchan.g*, the values of these constants were chosen for a granule cell simulation using Hodgkin–Huxley channels, but with potentials that are slightly shifted from the squid potentials which we defined in the last tutorial. Thus, it is important to place the “*include hhchan*” statement before our own definitions of these constants, so that ours will prevail. It is also a good idea to put in comments that explain the necessary order of these statements.

We can understand how these constants and the *SOMA_A* constant are used by looking at the definition of the function *make_Na_squid_hh*, which follows some extended comments describing the **hh_channel** object. The function begins by checking for the existence of an element called *Na_squid_hh*. If this channel doesn’t already exist, one is created and the internal fields are set with a lengthy *setfield* command that begins with

```
setfield Na_squid_hh \
    Ek           {ENA} \          // V
    Gbar        { 1.2e3 * SOMA_A } \ // S
```

The field *Ek* is the “battery” that is in series with the sodium conductance. It will be set to the value (in volts) that we have defined for the sodium equilibrium potential, *ENA*. In their paper, Hodgkin and Huxley (1952d) fit *Gbar* to a value of 120 milli-mhos (*mS*) per square centimeter of membrane area. You should verify that if we calculate *SOMA_A* in square meters, the expression above will correspond to the Hodgkin–Huxley result in SI units. In your script, use the values that we have given for the soma dimensions, *soma_l* and *soma_d*, to calculate *SOMA_A* for our cylindrical soma compartment. In doing so, you are letting this single Na channel represent the composite behavior of the many sodium channels that would be found in a patch of membrane with area *SOMA_A*. If you have the time (and inclination)

you may wish to verify that the values used to set the remaining fields correspond to those used by Hodgkin and Huxley. You should note, however, that the sign convention used for voltages by Hodgkin and Huxley is reversed from our modern usage.

The function *make_K_squid_hh* works in a similar manner to create a potassium channel called *K_squid.hh*. These two channels will be created at the current position in the hierarchical element tree. For a convenient grouping of elements, we would like to refer to these channels as */cell/soma/Na_squid_hh* and */cell/soma/K_squid_hh*. The easiest way to accomplish this is to use the *pushe* and *pope* commands to temporarily make */cell/soma* the current working element. We can create the two channels in the proper location by putting the statements

```
pushe /cell/soma  
make_Na_squid_hh  
make_K_squid_hh  
pope
```

in the main part of our script, after the statements that create the soma compartment.

At this stage, we now have the two channels, but we have not yet linked them to the soma. The soma needs to know the value of the channel conductance and its equilibrium potential in order to calculate the current through the channel. The soma will use this current as part of its calculation to update the soma membrane potential. The channel calculates its voltage- and time-dependent conductance using the current value of the soma membrane potential. As usual, these communication links are established by setting up messages between the elements. The soma may be linked to the sodium channel with the statements

```
addmsg /cell/soma/Na_squid_hh /cell/soma CHANNEL Gk Ek  
addmsg /cell/soma /cell/soma/Na_squid_hh VOLTAGE Vm
```

You should now add these to your script, along with the corresponding messages for the potassium channel. Before running the simulation, ask yourself if there is anything else that needs to be done.

14.4 Final Additions and Improvements

At the end of the previous tutorial, we discussed some guidelines for choosing an appropriate integration step to be specified with the GENESIS *setclock* command. With no active channels, this simulation produced a smooth asymptotic increase to a constant value of V_m . Now that we have added the channels, we expect to see action potentials with a fairly short rise time. You should therefore use either your knowledge of neurobiology or trial and error to set the step size to a suitable value.

You should now be able to click on the RUN button to produce a sequence of realistic looking action potentials. With the injection current set at 0.3 nA , they should occur at intervals of roughly 14 msec . You may notice something a little strange about the first few steps of the simulation, however. Instead of starting at a resting potential of -70 mV , the membrane potential starts at about -59 mV and then becomes more negative. This effect becomes more obvious if you set the injection current to zero. You would expect to see a constant V_m of -70 mV , but the first 10 msec of the simulation show a different behavior. In general, the initial steps of a neural simulation will not begin with the system in a “natural” state. One solution to this problem is simply to run the simulation until it reaches a steady-state behavior before taking data. In our case, it is possible to perform a more realistic initialization if we understand a few details of the GENESIS *reset* function and its effect upon **compartment** objects and channels.

14.4.1 Use of the Compartment *initVm* Field

In Sec. 12.5, we have discussed some of the actions performed by GENESIS objects. The *reset* command causes each element to perform its own RESET action. For example, this will cause an **xgraph** object to clear the graph, unless the *overlay* field flag is set. If the element is a **compartment** object, *reset* means that the membrane potential V_m will be initialized to the value of the *initVm* field of the compartment. We haven’t mentioned this field before, because it normally contains the same value as the *Em* field and follows any changes that are made to *Em*. Thus, the default behavior is that V_m gets initialized to *Em* after a reset. (Remember that *Em* is the “leakage” battery that is in series with the membrane resistance *Rm* in Figure 12.1.) The RESET action for the **hh_channel** object causes it to get V_m from any incoming messages and to use it to calculate initial values for the Hodgkin–Huxley rate constants and the channel conductance *Gk*. If there were no passive channels or sources of current other than the channels explicitly added to our model, we would set the soma *Em* field to *EREST_ACT* (-70 mV). As this would indirectly set the *initVm* field to the same value, this initial value of V_m would be used to calculate the initial channel conductances after a *reset*. These conductances, with the channel equilibrium potentials, would result in no net current flow into the soma. Thus V_m would remain at *EREST_ACT* in the absence of any current injection.

In the Hodgkin–Huxley model, the Na and K channels produce a net current flow at *EREST_ACT*. In order to offset this current, *Em* is set to *Eleak*, a leakage potential that differs from *EREST_ACT*. With the proper value of *Eleak*, there will be no net current flow, and V_m will remain at the steady-state value of *EREST_ACT*. However, after a *reset*, we want to start the simulation with V_m initialized to *EREST_ACT*, rather than *Eleak*. This may easily be accomplished by setting the soma *initVm* field to *EREST_ACT* after the soma compartment has been created and *Em* has been set to *Eleak*. Once *initVm* has been set to a value which is different from that of *Em*, future changes to *Em* will no longer affect *initVm*.

(If you later want to have *initVm* follow changes to *Em*, just set it to the same value as *Em*.) With this final addition to the script, everything will be properly initialized. Your simulation results should now look like those shown in Fig. 14.1.

You might notice that, even with this change, the first action potential is larger than subsequent ones. This is not a bug in the simulator or your script, but is a legitimate result of the Hodgkin–Huxley model. You may wish to experiment with different injection currents and see if you can explain this behavior.

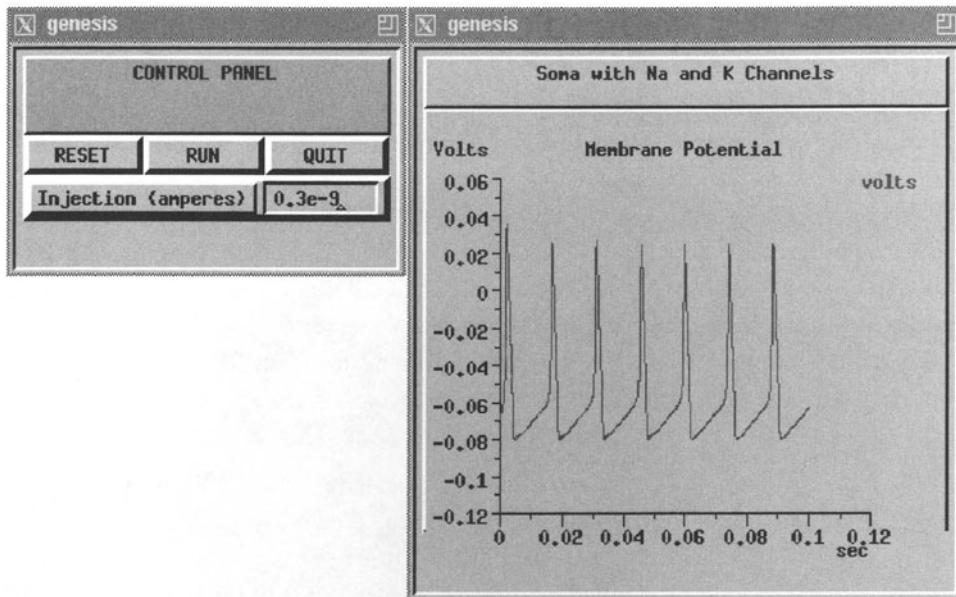


Figure 14.1 Typical results for the GENESIS simulation of a soma with Hodgkin–Huxley sodium and potassium channels.

14.4.2 Overlaying GENESIS Plots

There is yet another addition that you may make to your simulation if you choose. The command “`showfield /data/voltage -all`” reveals that your voltage graph has a field called *overlay* that was initialized to zero. As mentioned above, by setting it to a non-zero value, you may suppress the clearing of the graph during reset. This will allow you to overplot results using different injection currents, step sizes or other parameters. Although you can set this field to different values from the GENESIS command line, it would be nice to toggle it back and forth between zero and one by clicking on a button. As this lesson has been long enough, we will save the discussion of the XODUS toggle widget for the end of Chapter 15. If you would like to add toggle buttons to your simulations, you may find the information you need there, or in the GENESIS Reference Manual.

14.5 Extended Objects

You may encounter GENESIS scripts that use *extended objects* to create voltage-activated channels, rather than functions such as those defined in *hhchan.g*. Extended objects are created by a newly added GENESIS feature that allows you to use the GENESIS script language to create your own objects. The starting point for an extended object is an element or hierarchy of elements created from existing GENESIS objects that have at least some of the properties of the new object which you would like to create. New fields, message definitions and actions may then be added to the root element of the hierarchy before it is converted to an extended object.

For example, you may wish to create a specialized version of the **hh_channel** object in order to implement a Hodgkin–Huxley potassium channel. Or, you might replace the function *make_Vmgraph* with an extended graphical object that is appropriate for plotting membrane potentials. Typically, this would consist of a form, graph and overlay toggle button linked by messages, and would have the various fields pre-set to appropriate values for plotting membrane potentials. It might also be convenient to define a compartment object that can calculate and set its own values of *Rm*, *Cm* and *Ra* from the global variables *RM*, *CM* and *RA* using the compartment dimensions that have been set in its own *len* and *dia* fields.

As a specific example, let's create an extended object to represent a “squid-like” Hodgkin–Huxley potassium channel. When a channel element is created from this object, we would like it to not only have the fields for the various Hodgkin–Huxley constants (*X_alpha.FORM*, etc.) initialized to the appropriate values, but we would like it to perform some of the actions that were previously defined in our simulation script. For example, it should be able to calculate the area of its parent compartment from the compartment's *len* and *dia* fields, and use these to calculate and set *Gbar*, without having to previously specify a global variable *SOMA_A* and variables for the soma dimensions. It should also be capable of establishing its own messages with the parent compartment, so that we don't have to do this ourselves each time we add a channel to a compartment. We illustrate some of these features with excerpts from the script *hhchan_K.g*, which is listed in Appendix B.

The script begins much like *hhchan.g*, with the definition of some global constants, the creation of an **hh_channel** element called *K_squid_hh*, and the setting of the various fields. One difference is that a function is not defined to create *K_squid_hh*. Instead, it will be created by this script, modified, and then converted into a new GENESIS object with the same name. Also, we will arbitrarily set the *Gbar* field to zero, as it will be initialized to the proper value when the channel is created as the child element of a compartment.

Next, we add a new field *gdens* to the element to hold the conductance density of the channel, and set it to the default value of 360 S/m^2 . This is done with the statements

addfield	K_squid_hh	gdens	
setfield	K_squid_hh	gdens	360.0

The *addfield* command is used not just with extended objects, but may be used any time we wish to add a new field to an element.

As our new object will be a specific kind of channel, rather than a generalized Hodgkin–Huxley channel, it is best to protect the Hodgkin–Huxley channel constants from being inappropriately changed by the user. In fact, it would be a good idea to hide them from view, so that “*showfield -all*” will show only fields about which we care. We would like to be able to inspect the *Gbar* field, but it should only be indirectly settable by setting *gdens*. We set these two different levels of protection using the *setfieldprot* command with the statements

```
setfieldprot K_squid_hh -hidden Xpower Ypower X_alpha_FORM \
    X_alpha_A X_alpha_B X_alpha_V0 X_beta_FORM X_beta_A \
    X_beta_B X_beta_V0 Y_alpha_FORM Y_alpha_A Y_alpha_B \
    Y_alpha_V0 Y_beta_FORM Y_beta_A Y_beta_B Y_beta_V0
setfieldprot K_squid_hh -readonly Gbar
```

The second statement protects the *Gbar* field, but uses the “*-readonly*” option instead of “*-hidden*”, so that it will be visible with the *showfield* command, but will not be settable with *setfield*.

Now we need to define a function that extends the SET action of the **hh_channel** object to also set the *Gbar* field whenever the *gdens* field is set. This is accomplished with:

```
function K_squid_hh_SET(action, field, newvalue)
    float newvalue
    float PI = 3.14159
    if (field == "gdens")
        setfield . Gbar {PI*{getfield .. dia}*{getfield .. len}*newvalue}
    end
    return 0           // indicate that SET action isn't yet complete
end
```

When the *setfield* command is given, the three arguments *action*, *field* and *newvalue* are automatically passed as strings. As we would like *newvalue* (the value used to set the *gdens* field) to be a float, we redeclare it here.

This is the first time that we have mentioned the GENESIS *if* construct so far, although it also appears in *hhchan.g*. In the next chapter (Sec. 15.3) we will see an example of an *if-else*. In either of these constructs, the word “*if*” is followed by a space and then a pair of parentheses that contains a logical expression. If the expression is true, it evaluates to 1 and the statements preceding “*end*” will be executed. Note the use of the two equals signs

(“==”) to represent the logical operator for “equal to,” as distinct from the single sign used to assign a value. The GENESIS logical operators are similar to those used in C, and are described in the GENESIS Reference Manual.

In our case, we want our new SET action to do something special if the field being set is our newly added *gdens* field. Otherwise, the default SET action of the **hh_channel** object will take over. Specifically, we want to multiply the new value of *gdens* by the area of the parent compartment and use this to set the *Gbar* field of the channel. Fortunately, the **compartment** object has two fields, *len* and *dia*, which may be used to hold the length and diameter of the compartment. Note that the “.” in the *setfield* expression refers to the working element (our channel) and that the “..” in the two *getfield* expressions refers to its parent element (the compartment containing the channel). When the SET action is called, the channel element becomes the working element during execution of the *K_squid_hh_SET* function. This change of working element takes place for all action functions. Also note that it is not necessary to name the working element with *setfield* and *getfield* commands, so the “.” could have been omitted. We also need to be sure that the *gdens* field itself gets set to *newvalue*. This will be done by the default SET action if we indicate that our SET action function did not set the field directly. We provide this information with the statement “return 0”, which causes the function to return a value of zero.

Having defined a function to implement the new SET action, we next use the *addaction* command

```
addaction K_squid_hh SET K_squid_hh_SET
```

to specify the name of the element (*K_squid_hh*), the name of the action that it is to perform (SET) and the name of the function that implements the action (*K_squid_hh_SET*).

This will properly set *Gbar* when we change the value of *gdens*, but we would like to properly initialize *Gbar* when the element is first created. We would also like the messages that link the channel to its parent compartment to be automatically added when the element is created. Finally, we should put in some protection to ensure that our new kind of channel can only be created as a child element of a compartment.

We can accomplish these with a similar function that extends the CREATE action, and with an appropriate *addaction* command:

```
function K_squid_hh_CREATE(action, parent, object, elm)
    float PI = 3.14159
    if (!{isa compartment ..})
        echo K_squid_hh must be the child of a compartment
        return 0
    end
    setfield . Gbar \
        {PI*{getfield .. dia}*{getfield .. len}*{getfield . gdens}}
```

```

addmsg . . . CHANNEL Gk Ek
addmsg . . . VOLTAGE Vm
return 1
end
addaction K_squid_hh CREATE K_squid_hh_CREATE

```

Here, we use the *isa* command to see if the parent element is derived from the **compartment** object. The negation operator “!” is then used so that if the channel’s parent is not a compartment, an error message will be given and the function will return a zero, indicating that creation of the element failed. If the test is passed, we go on to calculate and set *Gbar* using the compartment dimensions and the default value that we previously established for *gdens*. The two *addmsg* commands are similar to those at the end of Sec. 14.3, adding a CHANNEL message from the channel to the compartment, and a VOLTAGE message from the compartment to the channel. The function ends with “*return 1*” to indicate that it was completed without error.

Finally, we want to turn the *K_squid_hh* element into an object called **K_squid.hh**. This is done with the *addobject* command, which is followed by the name of the object to be created and then the name of the element from which it was made. This command also allows options for specifying the name of the object’s author and a description of the object. These will then appear with *showobject*, just as with a built-in GENESIS object. In our example, we have:

```

addobject K_squid_hh K_squid_hh -author "J. R. Hacker" \
          -description "Hodgkin--Huxley Active K Squid Channel - SI units"

```

Now, let’s test the script. Start GENESIS, change to the *Scripts/tutorials* directory where *hhchan_K.g* should be found, and give the following commands:

```

hhchan_K
le
listobjects

```

Note that the element *K_squid_hh* doesn’t exist, but that there is a new object of this name. Of course, we could have used a different name for the basis element and the extended object that we created from it. Either way, the *addobject* command destroys the original element when the new object is created. Try the commands

```

showobject K_squid_hh | more
showobject hh_channel | more

```

What differences do you note? Now try

```

create compartment /compt
setfield /compt dia 30e-6 len 30e-6
create K_squid_hh /compt/K
showfield /compt/K -all
showmsg /compt/K

```

How do the results of the *showfield* and *showmsg* commands compare with those obtained for the */cell/soma/K_squid_hh* element in your tutorial script? (You can also load *tutorial3.g* without interfering with any of the commands given so far.)

Try setting the */compt/K gdens* field to a different value and inspecting the value of *Gbar*. What happens if you try to set *Gbar* directly? What happens if you try to create a *K_squid_hh* element as a child of an element that is not a compartment?

The GENESIS Reference Manual describes some other capabilities of extended objects that are useful when constructing a compound object from a hierarchy of elements. Normally, all access to elements of the new object are via the fields, messages and actions of the root element, and the child elements are not accessible. Although this is usually desirable, there are cases when we want to access some of the fields and messages of one of the child elements. For example, if the root element is the form that contains the graph for plotting membrane potential, we would want to be able to add PLOT messages to the graph and to set the fields that specify the ranges for its axes. This may be done with *message forwarding* and *indirect fields*.

The GENESIS Reference Manual also describes the procedure for defining new objects and GENESIS commands by programming them in C, compiling them and then linking them into a new version of the executable *genesis* file. If this procedure is used to create a new object that has a computationally intensive PROCESS action added, it will execute more rapidly than one that is created at the script level. Nevertheless, it may speed development to make initial tests using an extended object as a prototype. For the example we have given here, there is little speed disadvantage in using an extended object, because the script language functions are used only when an element is created, or when fields are set. The compiled code for the original **hh_channel** object is used during the PROCESS action.

14.6 Exercises

1. It would be useful to be able to modify both the time step *dt* and the run time *tmax* from dialog boxes. Add these features to your simulation and have the x-axis of the graph adjust to the new *tmax*.
2. The function *make_Ca_hip_traub91* in *neurokit/prototypes/traub91chan.g* will create a high threshold calcium channel *Ca_hip_traub91*. (Except for the more descriptive name, this is the same as the Ca channel used in the *traub91* tutorial of Chapter 7.)

Chapter 19 describes how this channel was created from a **tabchannel** object. As with the channels in *hhchan.g*, we may use it without having to be concerned with the details of the *traub91chan.g* script. Copy this file into your directory and *include* it in a copy of your simulation script. Then make the changes necessary to add the channel to your model. In order for it to have a significant effect, you will need to increase the value of the *Gbar* field from its default value. What value of *Gbar* gives an appreciable “shoulder” to the action potentials?

3. In Sec. 14.2, we mentioned the *makegraphscale* function. Use this function to add a **scale** button to each of your graphs, so that you may easily change the *xmin*, *xmax*, *ymin*, and *xmax* fields.
4. The *Squid* tutorial (Chapter 4) uses the script *squid_electronics.g* to simulate the circuitry that is used for voltage clamp experiments. This is accomplished with a combination of the **PID** (Proportional, Integral, Derivative controller), **diffamp** (differential amplifier), and **RC** (low pass filter) objects. The GENESIS Reference Manual and the on-line help provide additional documentation for these devices. Adapt this script to provide voltage clamp capability for your simulation.
5. Why is the first action potential of the series shown in Fig. 14.1 higher than the following ones? It may help to revisit the *Squid* tutorial (Chapter 4) in order to examine plots of some of the other variables.
6. Make a script analogous to *hhchan_K.g* to create an extended object for the Na channel. Modify your *tutorial3.g* script to use extended objects instead of the functions defined in *hhchan.g*. Don’t forget to set the *len* and *dia* fields of the soma compartment.

Chapter 15

Adding Dendrites and Synapses

DAVID BEEMAN

In the previous chapter's tutorial, we created a single soma compartment with Hodgkin–Huxley sodium and potassium channels. Your script for this simulation should look something like the one given in the listing of *tutorial3.g* in Appendix B. In this simulation, we will build upon this script in order to construct a multi-compartmental neuron with a dendrite compartment containing a synaptically activated channel, an active soma, and an axon.

15.1 Adding a Dendrite Compartment

We will start by making a single dendrite compartment that we will connect to the soma. For a more detailed model of a single neuron, we might have many such compartments linked together, possibly with much branching. In this case, it would be advisable to use one of GENESIS' implicit numerical integration methods that are described in Chapter 20, in order to avoid the need for very small time steps. However, for our two-compartment model, and for models containing only a few compartments, the default method used by GENESIS will provide sufficient numerical accuracy with a moderate integration step size.

An appropriate size for the compartment would be $100 \mu m$ long and $2 \mu m$ in diameter, so we add the following definitions to our script after the definitions of the soma dimensions:

```
float dend_l = 100e-6 // add a 100 micron long dendrite
float dend_d = 2e-6 // give it a 2 micron diameter
```

You may wish to verify that these dimensions are consistent with the criterion that the compartment length should be small compared to the electrotonic length of the compartment (Segev, Fleshman and Burke 1989). We can make the dendrite compartment with the same *makecompartment* function that was used to create the soma, giving it the name */cell/dend* and using the dendrite dimensions instead of the soma dimensions. As the dendrite compartment will not have Hodgkin–Huxley channels, the field *Em* should be set to *EREST_ACT*, rather than *E leak*. Our cell is now becoming complex enough to merit a *makeneuron* function of its own. The main script should create the cell with the function call

```
makeneuron /cell {soma_1} {soma_d} {dend_1} {dend_d}
```

and the statements that make the dendrite compartment and the soma with its channels should go into the definition of this function. As it will make use of the *makecompartment* function, its definition will have to come after that of *makecompartment*.

As we will generally stimulate the neuron with input to the dendrite rather than with current injection, let's now set the soma injection field value to zero instead of using the previous value of 0.3×10^{-9} amperes. This should be done in the main body of the script, after the call to *makeneuron*. For consistency, the initial value of the dialog box in the function *make_control* should also be set to zero.

Now we need to link the dendrite to the soma. In Fig. 2.2, the dendrite compartment would correspond to the “primed” compartment shown at the left. The dendrite compartment needs to send both its axial resistance and its membrane potential at the previous simulation step to the soma compartment. This allows the soma to calculate the current entering from the dendrite compartment. This is done in the first message below, where the dendrite compartment is linked to the soma with a message of the type RAXIAL. This message has two value fields, *Ra* and *previous_state*. The *previous_state* field gives the value of the membrane potential *Vm* at the previous integration step. We use this field rather than *Vm* because GENESIS updates the fields of all the compartments in parallel, and we want each compartment to update its data fields using data from the previous simulation step.

As the dendrite knows its own axial resistance to the soma, it only needs to receive the soma's previous membrane potential in order to update its state. This is accomplished with the second message, which is named AXIAL. If we were to use the variable *path* as the name of the parameter that will be replaced by */cell* when *makeneuron* is invoked, the statements that set up the messages would be

```
addmsg {path}/dend {path}/soma RAXIAL Ra previous_state
addmsg {path}/soma {path}/dend AXIAL previous_state
```

Now we need to add a synaptically activated channel to the dendrite compartment. The GENESIS object most suitable for this is the **synchan** object. This object was used to create

the synaptically activated channels in the *Neuron* tutorial, which we used in Chapter 6. Some older simulation scripts make use of a similar GENESIS object, the **channelC2**, which is used with the obsolete **axon** object.

The **synchan** may receive delta-function “spike events,” each lasting for a single integration time step, from a SPIKE message. It then calculates a net channel conductance G_k summing the effects of each spike. The parameter fields $gmax$, $tau1$ and $tau2$ are used to determine the time behavior of the conductance. When the $tau1$ and $tau2$ fields are equal, a single spike impulse gives a conductance with the time dependence

$$G_k = gmax \frac{t}{tau1} \exp\left(1 - \frac{t}{tau1}\right). \quad (15.1)$$

This causes G_k to reach a maximum value of $gmax$ after a time $tau1$. The initial rise in conductance is linear and the decay is exponential, with time constant $tau1$. This is the “alpha function” form, corresponding to Eq. 6.16.

When the two time constants differ, the conductance assumes a dual exponential form,

$$G_k = \frac{A gmax}{tau1 - tau2} \left(\exp\left(-\frac{t}{tau1}\right) - \exp\left(-\frac{t}{tau2}\right) \right), \quad (15.2)$$

where A is a normalization constant chosen so that G_k assumes a maximum value of $gmax$. These fields correspond to the variables that appear in Eq. 6.17. As with voltage-activated channels, there is a field E_k for the equilibrium potential of the channel. You may find further information about this object with the GENESIS command “help synchan”. The mathematics behind the implementation of the **synchan** object are described by Wilson and Bower (1989).

It may be useful to add other channels later, so it would be a good idea to write a fairly general function that lets us specify arguments for the path to the compartment, the name of the channel, and the values of the four parameters E_k , $tau1$, $tau2$, and $gmax$. You might start the declaration with:

```
function makechannel(compartment,channel,Ek,tau1,tau2,gmax)
```

The body of the function should be written so that the statement

```
makechannel /cell/dend Ex_channel {Ek} {tau1} {tau2} {gmax}
```

will create a channel named */cell/dend/Ex_channel* and will set the fields to the stated values.

The function also needs to link the channel to its parent compartment. As usual, this is done by passing messages. The compartment needs to know the conductance of the channel and the equilibrium potential of the channel (the voltage of the “battery” in series with the conductance). This is used by the compartment in its calculation of the net current flow into the compartment. Although the channel conductance is not dependent on the membrane

potential, the **synchan** object also calculates the channel current, so it needs to receive a message from the compartment that gives the membrane potential. The required messages may be set up with the statements, similar to the ones used with the Hodgkin–Huxley channels in the soma,

```
addmsg  {compartment}/{channel}  {compartment} CHANNEL Gk Ek
addmsg  {compartment}  {compartment}/{channel} VOLTAGE Vm
```

Use this function within *makeneuron* to create an excitatory channel having a sodium equilibrium constant, both time constants equal to 0.003 second, and a *gmax* of 5×10^{-10} siemens.

In order to do anything interesting with our neuron, we will need to give it some synaptic input to the dendrite excitatory channel. However, before continuing, it would be a good idea to test what we have so far. Try running the script as it exists at this stage, and use the dialog box to give the soma an injection current. Does it still work as before? Although there is no dialog box for injection current to the dendrite, you can give it some injection by typing the command “*setfield /cell/dend inject 0.3e-9*” to the GENESIS prompt. With no injection to the soma, does injection to the dendrite still produce action potentials in the soma? Can you detect any difference in the results when injecting the dendrite rather than the soma?

15.2 Providing Synaptic Input

The GENESIS Reference Manual describes various methods of activating a **synchan** element. The most common method is to convert the action potentials produced in another neuron to a sequence of unit amplitude spikes that are sent to the **synchan** with a SPIKE message. As we don’t have another neuron to provide synaptic input to */cell/dend/Ex_channel*, it would be nice to have a function that would provide a randomly distributed train of spikes to a given channel. This might represent the random spontaneous “background level” firing of many other neurons that have inputs to our cell. Let’s call the function *makeinput(path)*, so that “*makeinput /cell/dend/Ex_channel*” will provide the necessary input. GENESIS has a number of objects that can be used to generate trains of pulses. We will use the **randomspike** object to create an element that produces random spikes. Let’s call this element */randomspike*, also.

The **randomspike** object has the settable parameter fields, *rate*, *min_amp*, *max_amp*, *reset*, and *reset_value*. The first of these gives the average number of randomly generated spike events per unit time. When an event has been generated, the amplitude of the event is a random variable uniformly distributed between *min_amp* and *max_amp*. There is also a field named *state* which is updated at every simulation step. The *state* field has the value of the event amplitude if an event has been generated. If an event is not generated, then the

value of the *state* field depends on the *reset* field. If *reset* is non-zero, then *state* takes on the value given in *reset_value*. Otherwise *state* will behave like a latch containing the amplitude of the previous event.

An average of twenty spikes per 0.1 second run of the simulation would be a good spiking rate to use in order to represent the input from several other neurons. We would like them all to be of unit amplitude, and to last for just a single time step. After */randomspike* is created, this can be accomplished with the statement

```
setfield ^ min_amp 1.0 max_amp 1.0 rate 200 reset 1 reset_value 0
```

We can then send the spike train to the channel with

```
addmsg /randomspike /cell/dend/Ex_channel SPIKE
```

Note that the SPIKE message uses no parameters. Whenever a spike event is produced, */randomspike* notifies the channel of the event. The **synchan** regards this as a spike of unit amplitude, without making use of the *state* field of the **randomspike** object.

A typical axonal connection from another neuron involves a propagation delay. This can be calculated from the axonal propagation velocity (on the order of 1 meter per second) and the length of the axon. Typical values of this delay are in the range of 1 to 10 msec. In addition, there can be a time lag of slightly less than 1 msec between the arrival of a presynaptic event and the postsynaptic response. Also, we may need to scale a large network down to a smaller model. This means that a single synaptic connection in our model might represent several similar inputs in the biological system. Therefore, we would like a way to scale the effect of a single synaptic connection by a “weight” factor as we did in the *Neuron* tutorial experiments described in Sec. 6.5.1.

Each synaptic connection that is established by the addition of a SPIKE message contains a field for the total delay and for the synaptic weight. In this particular case, we don’t care about the delay, and the spike rate is fast enough to provide a reasonable amount of input for a single synapse. The synaptic connections are numbered starting with zero, so we would set these fields with the statement

```
setfield /cell/dend/Ex_channel synapse[0].delay 0 synapse[0].weight 1
```

It would be useful to plot the output of */randomspike* on the same graph as we use for the soma *V_m*. This could be done by sending a “PLOT state” message to the graph. However, we have a problem with the very different magnitudes of *V_m* and the state of *randomspike*. Fortunately, the **xgraph** object can receive a message called PLOTSCALE which allows one to specify a scale factor and offset for the field to be plotted, in addition to the label and color. The syntax for this message is

```
addmsg source_element dest_graph \
    PLOTSCALE field *label *color scale offset
```

Use this message to plot the spikes, giving them a height of 0.01. After having made these additions to your script, try it out. Are the results reasonable?

A plot of the conductance of */cell/dend/Ex_channel* can help our understanding of the conditions under which action potentials are generated in the soma. There should be room for such a graph to the right of the graph */data/voltage*. Use your understanding of the function *make_Vmgraph* to make an analogous function to create a graph *channel_Gk* within the form */condgraphs*. Set the vertical scale (*ymax*) to $10gmax$. Notice the effect of each spike on the channel conductance, and the relationship of the conductance to the production of action potentials.

You should be aware of the fact that we have done some things the hard way in this section in order to illustrate some GENESIS features. For example, the use of the PLOTSCALE message to the graph was not strictly necessary. As the **synchan** object uses a SPIKE message only to detect the existence of a spike event, we are free to set the *min_amp* and *max_amp* fields of */randomspike* to values that are convenient for plotting. If we had been interested only in randomly activating the channel, we could have done it much more easily by setting the **synchan** *frequency* field to the desired frequency of random activation. You may wish to try deleting the SPIKE message (or deleting */randomspike*) and setting the *frequency* field to 200.

15.3 Connections Between Neurons

Normally, our cell would have a synaptic connection to another neuron, as in the *MultiCell* demonstration. In order to see how this would be done, we can provide a positive feedback connection from our cell to itself.

If we were interested in the details of the propagation of action potentials along an axon, we might wish to build a multi-compartmental model of an axon connected to the soma. For most purposes, we can use a much simpler model of an axon, regarding it as a simple delay line for the propagation of spikes. As we have described, the axonal propagation delay is combined with the synaptic latency and is implemented within the **synchan** object.

Typically, a presynaptic terminal releases a quantity of neurotransmitter near the peak of an action potential. This means that we can achieve some computational efficiency by converting each action potential to a single spike before it is sent to the **synchan**, instead of explicitly calculating the postsynaptic response to the time-varying presynaptic membrane potential. (When it is necessary to model the postsynaptic response to a graded non-spiking potential, an ACTIVATION message may be sent to the **synchan**.) We can accomplish this conversion by linking the soma to a **spikegen** object with an INPUT message. The command

“`help spikegen | more`” reveals that this object sets its *state* field to a value *output_amp* for a single time step. This occurs whenever it receives an input greater than the value of the field *thresh* and there has not been a spike for at least the interval specified by *abs_refract*. As the action potentials are quite steep near $V_m = 0$, this would be a good value to use for the *thresh* field of the spike element.

During an action potential, V_m will generally be above the threshold for longer than a single time step. As we want each action potential to generate a single spike, we also need to set the *abs_refract* field to a value corresponding to the minimum expected interval between action potentials. A typical refractory period would be on the order of 0.01 seconds. As with the */randomspike* element, we would like the spikes to be of unit amplitude. The necessary statements would be of the form

```
create spikegen /cell/soma/spike
setfield /cell/soma/spike thresh 0 abs_refract 0.010 output_amp 1
addmsg /cell/soma /cell/soma/spike INPUT Vm
```

At this point, you should add similar statements to *makeneuron* that will add a **spikegen** element to the soma, with the name of the cell being specified by the *path* argument.

Once the cell has been created, we can use a SPIKE message to establish a synaptic connection from */cell/soma/spike* to */cell/dend/Ex_channel*, just as for the connection from */randomspike*. As a single synaptic input to a dendrite is generally not sufficient to excite a neuron, we will weight the input by a factor of 10, as if the cell were receiving inputs from ten identical synapses. We will also give the input a propagation delay of 5 msec. Try this out, adding the necessary statements to your simulation script. Remember that this second connection will be numbered as *synapse[1]*.

As a final embellishment to our simulation, we can add a “button” to the conductance graph form that toggles this feedback connection on and off. The **xtoggle** widget was mentioned in Chapter 14 as a possible way to toggle the *overlay* flag field of an **xgraph** object. This object is similar to the **xbutton** object, with many of the same fields and options. In addition, the field *state*, which may be accessed with the GENESIS *getfield* function, toggles back and forth between 0 and 1 when the toggle button is clicked with the left mouse button. It also has the fields *offlabel* and *onlabel*. These may be set to the two strings that will be displayed when the toggle state is 0 or 1. As usual, the *-script* option can be used to specify the name of a function to be invoked when the toggle is clicked. Typically, this function will inspect the state of the toggle and use an *if-else* construct to perform the appropriate operation. In our case, the function might look like this:

```
function toggle_feedback
    int msgnum
    if ({getfield /control/feedback state} == 0)
```

```

        deletemsg /cell/soma/spike 0 -out
        echo "Feedback connection deleted"
    else
        addmsg /cell/soma/spike /cell/dend/Ex_channel SPIKE
        msgnum = {getfield /cell/dend/Ex_channel nsynapses} - 1
        setfield /cell/dend/Ex_channel \
            synapse[{msgnum}].weight 10 synapse[{msgnum}].delay 0.005
        echo "Feedback connection added"
    end
end

```

In this function, we have made use of the *nsynapses* field of the **synchan** object. As it is often easy to lose track of the number of the most recently created connection, we can use *nsynapses* to find the number of SPIKE messages that exist. It is then decremented by one to take into account the fact that the first synapse is number 0.

Create an **xtoggle** element with appropriate labels in the */control* form and use this function to allow it to toggle the feedback connection on and off. If you have done everything correctly, your simulation results should resemble those in Fig. 15.1, when the feedback connection is “off.”

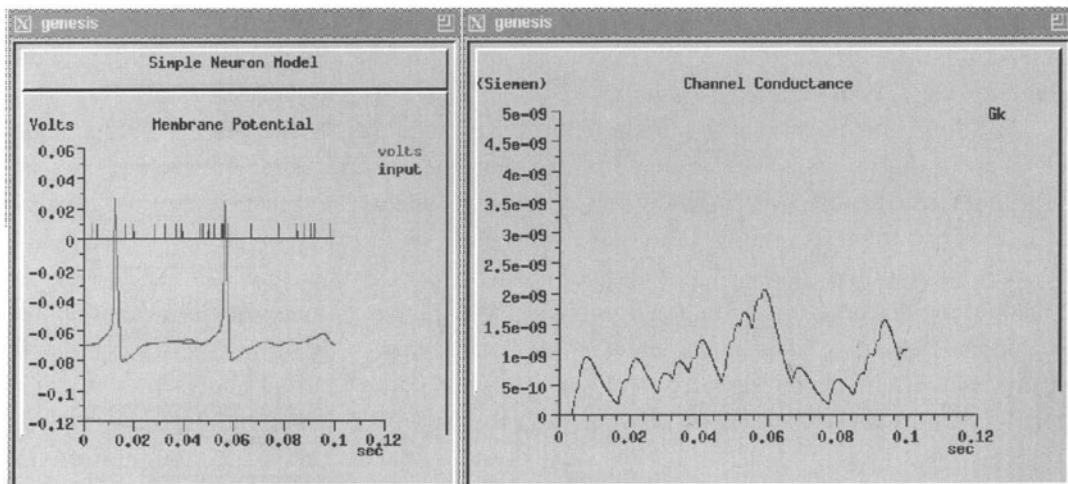


Figure 15.1 Typical results for the simulation when the dendrite excitatory channel is stimulated with random spike events. The feedback connection from the axon has been toggled “off.”

15.4 Learning and Synaptic Plasticity

To implement learning or other forms of adaptive behavior in a GENESIS simulation, we need some way to modify the synaptic weight, or to otherwise change the effect of providing synaptic input. The following sections describe some of the ways that this can be done.

15.4.1 Continous Modification of the Synaptic Weight

When implementing learning algorithms, you will likely want to modify specific connection weights, as with:

```
setfield /cell/dend/Ex_channel synapse[0].weight {new_weight}
```

However, you will want to make these changes continuously, with changing values of the variable *new_weight*, while the simulation is being stepped.

This could be done with a function written in the GENESIS script language. There is a GENESIS object called **script.out** that could be used to invoke this function at specified intervals during the simulation. Alternatively, you could use a **synchan** as the basis for an extended object that performs some weight-changing algorithm as part of its PROCESS action.

15.4.2 Use of the MOD Message

The **synchan** is able to receive a MOD message, which is intended for implementing neuro-modulation, but could also be used to cause learned time-dependent modification of synaptic activation. This message simply scales the channel activation for the current time step by a factor that is sent with the MOD message. Note that this globally affects all the synapses in the **synchan**. If you want some synapses to be modifiable, but not others, you should divide your channel into two **synchans** and send a MOD message only to the modifiable one. As with the method of directly setting the synaptic weight fields, you would most likely use a **script.out** or extended object to provide the modification algorithm and calculate a value to be sent to the **synchan** with the MOD message.

15.4.3 Hebbian Learning with the hebbsynchan

Hebb (1949) postulated a simple rule for learning that was based on a correlation of the presynaptic and postsynaptic activity of a neuron. More recently, Hebb's rule has been applied to the understanding of the basis of long term potentiation (LTP), a persistent increase in synaptic efficiency that can be rapidly induced. Brown, Kairiss and Keenan (1990) have given a detailed review of these Hebbian synapses and the biophysical mechanisms that

underlie their behavior. A modern definition of a Hebbian synapse defines it as one that increases its strength with correlated pre- and postsynaptic activity, and decreases its strength with negatively correlated activity. An anti-Hebbian synapse modifies its synaptic strength by rewarding negatively correlated activity and punishing correlated activity.

The **hebbsynchan** object provides for both Hebbian and anti-Hebbian modification of the weight field of a synaptic connection. This object, described in the GENESIS Reference Manual, is very similar to the **synchan**, except that the synaptic weights are not fixed, but vary as a function of both the pre- and postsynaptic activities. In addition to the *weight* and *delay* fields, these synapses have a field called *pre_activity* which represents an averaging of the presynaptic spiking activity through that synapse. Note that each synapse has its own *pre_activity* field, just as each synapse has its own field for its weight and delay. The postsynaptic activity is the same for all synapses in the **hebbsynchan**, and is a function of the averaged membrane potential of the compartment to which the **hebbsynchan** is connected. The *Scripts/examples/hebb* directory contains a demonstration based on the script that was developed in this chapter, but which uses the **hebbsynchan** instead of the **synchan**.

15.4.4 Customizing the **synchan** or **hebbsynchan**

If your learning algorithm does not fall into a category that is implemented by the **hebbsynchan**, you should consider writing your own customized synaptic channel object in C, to be compiled into the simulator. Although it may be helpful to use **script_out** or extended objects for initial development and testing of your learning model, your simulation will run faster if you use compiled objects. The GENESIS Reference Manual chapter “Customizing GENESIS” provides detailed instructions for adding new objects and commands to GENESIS. If your new object is based on a modification of the **synchan** or **hebbsynchan**, the section “Creating New Synaptic Objects” will be particularly useful. In order to make modification of the weight change algorithm for the **hebbsynchan** easy, it has been isolated to a single function in the source file *hebbsynchan.c*.

15.5 Where Do We Go from Here?

At this point there are a number of directions to go for learning more about GENESIS programming. The demonstration simulation *MultiCell* connects two neurons such as we have created here in an excitatory-inhibitory loop to produce bursts of pulses. The accompanying file *MultiCell.doc* gives detailed commentary on the syntax of the scripts that are used in the simulation. You may wish to modify your script for this tutorial to create a second neuron and produce your own version of *MultiCell*. It would also be a good idea to study the scripts for the *CPG* simulation from Chapter 8. The *Neuron* tutorial scripts have good examples of functions for providing short trains of spikes as inputs to a synapse.

For learning how to copy a cell into a large array of interconnected cells, the *Orient_tut* demonstration (discussed in Chapter 18) is worthy of study. This simulation also demonstrates some advanced XODUS features using the “draw” (**xdraw**) widget. Some of these are discussed in Chapter 22.

The next two chapters use the GENESIS *cell reader* to create this same cell with a few concise commands by reading a data file. The use of the cell reader is the preferred method for constructing complex neurons with many compartments and channels.

15.6 Exercises

1. Examine the scripts for the *MultiCell* simulation and determine the parameters that were used for the two neurons and the mutual connections between them. Use your *makeneuron* function to create the two cells and then provide synaptic connections with the same characteristics as those in *MultiCell*. Demonstrate that the firing patterns are the same.
2. Try gradually reducing the amount of delay in the feedback connection used in our model. Why does this eventually *increase* the interval between action potentials?
3. At the end of Chapter 14, we mentioned the *overlay* field of the **xgraph** object. Add a toggle button to each of your graphs so that you can switch back and forth between overlay mode.
4. When we connected the soma compartment to the dendrite compartment, we set up the messages so that the membrane potentials of the two compartments were connected through the dendrite’s axial resistance. Make another copy of your simulation script that connects the two compartments through the soma’s axial resistance. Use the *showmsg* command to verify that the connections are really different.

Then modify both versions of the simulation so that there are no channels, and provide 0.3 nA current injection to the dendrite compartment, instead of to the soma. (Rather than modifying the simulation script, you may find it easiest to delete the channels once the simulation is loaded, and to set the dendrite *inject* field from the GENESIS prompt.)

Explain the differences between the results that you obtain for the two situations. What happens if you significantly reduce the simulation time step? (You may either use *setclock* for this, or use a dialog box like the one described in Exercise 1 of Chapter 14.) How small does it need to be in order to get accurate results when the two compartments are connected through the soma’s axial resistance? Explain this result in terms of the relevant time constants that arise from the resistances and capacitances

in this model. When making a realistic neural model, why is it conceptually wrong to connect the compartments through the soma Ra ?

Chapter 16

Automating Cell Construction with the Cell Reader

DAVID BEEMAN

16.1 Introduction

In the previous chapter, we created a simple multi-compartmental neuron with a dendrite compartment, a soma, and an axon. The dendrite contained a synaptically activated channel and the soma contained voltage-activated Hodgkin–Huxley sodium and potassium channels. The script *tutorial4.g*, listed in Appendix B, contains the function definitions and commands used to construct this neuron, provide synaptic input to the cell, and plot the results of the simulation. In this tutorial, we will modify the script in order to construct the same neuron from a data file, using the GENESIS cell reader. The cell reader, which is implemented in the *readcell* command, allows one to build multi-compartmental neurons by reading cell parameters from a *cell descriptor file*. This file contains the names and dimensions of the compartments that will be used in the cell, along with the names of the channels and other elements that are contained within each compartment. The cell reader then uses this information to put together a cell and to establish the necessary messages between the various elements. This can significantly reduce the effort needed to construct a complex cell with many compartments and channels.

Before reading further, you may wish to review Chapter 15 and the listing of *tutorial4.g* in order to remind yourself of the purpose of the various functions that were defined in the simulation script. In this tutorial, we will use seven scripts to create the simulation. The

following section describes the use of a script that we will write (*protodefs.g*) in order to create a library of the basic components to be used in building our cell. This script may use *include* to bring in other scripts that define the particular channels or other elements which we need. In our case, we will use the *hhchan.g* script that we used in the previous chapters, plus three other scripts from the *neurokit/prototypes* directory (*compartments.g*, *synchans.g* and *protoSPIKE.g*). Then, we describe the listing of the cell descriptor file *cell.p*. Finally, we will construct a main simulation script *tutorial5.g*. This will create the various XODUS elements needed to control and display the simulation, include *protodefs.g* in order to create the raw materials for building the cell, process *cell.p* with *readcell* to create the cell, and create any needed external objects and their messages in order to interact with the cell.

16.2 Creating a Library of Prototype Elements

The cell reader builds the cell by making copies of “prototypes” of the various elements that will be used, replacing the default values of parameter fields with values taken from the cell descriptor file. For example, when constructing a soma with several attached dendrite compartments, it will make multiple copies of a generic compartment prototype and then set the data fields in each compartment to the appropriate values. Likewise, a cell having Hodgkin–Huxley Na channels in several compartments will get these channels from copies of the prototype. These channels will be identical, but the cell reader will provide the right value of the maximum channel conductance *Gbar* for each copy.

The cell reader expects to find this library of prototype elements as a set of subelements of the **neutral** element */library*. Thus, we need to write a script that will create */library* and fill it with a prototype compartment, one copy of each of the different channel types we will use, and a spike generator. Although the statements that are needed to set up the prototype library could go into your main simulation script, it is customary to make a separate script for this, and to then use *include* to bring it into the simulation. Following tradition, we will call this script *protodefs.g*, although you may give it any name you like.

16.2.1 Future Changes in the Cell Reader

The cell reader is continually evolving in order to provide more flexibility. In future releases of GENESIS, more options will be available for the cell descriptor file and new types of GENESIS objects may be used as components for the construction of cells. In order to be aware of these new developments, you should consult the current version of the *readcell* documentation in the GENESIS Reference Manual.

It is likely that future versions of the cell reader will construct cells from extended objects instead of the */library* prototype elements we have used here. As we have demonstrated in Sec. 14.5, extended objects can take over many of the duties that are performed by

script commands or by the cell reader, such as establishing messages with their parent compartments and scaling various fields according to the compartment dimensions. This will mean some changes in the way that prototypes are specified in the *protodefs.g* file. The scripts that are included will typically create extended objects for prototype compartments, channels and other cell components, rather than providing functions that create elements from the basic predefined GENESIS objects. As these changes are incorporated into future releases of GENESIS, they will be accompanied by documentation and example scripts.

Instead of directly creating prototype elements from the basic **compartment**, **synchan** and **spikegen** objects, we will create them from functions defined in the scripts *compartments.g*, *synchans.g* and *protospike.g* in the *neurokit/prototypes* directory. Under GENESIS version 2, this is not strictly necessary. However, by including these scripts, we can maintain compatibility with future releases of GENESIS that may have modified versions of these scripts to go along with changes in *readcell*.

16.2.2 The *protodefs.g* Script

We begin by including the script *compartments.g*, which can create a variety of compartments. As in *tutorial4.g*, we can include the file *hhchan.g* in order to provide the functions to create the prototype Hodgkin–Huxley channels *Na_squid.hh* and *K_squid.hh*. We will also include the scripts *synchans.g*, which contains a function to create a glutamate-activated excitatory channel like the *Ex_channel* element we have used before, and *protospike.g*, which can create a **spikegen** element. If the SIMPATH in your *.simrc* file is properly set to include the *Scripts/neurokit/prototypes* directory, these files need not exist in your own directory. After including the *hhchan.g* file, be sure to replace its values for *EREST_ACT*, *ENA*, and *EK* with your own, as we did in *tutorial4.g*. As the cell reader will calculate the compartment area from dimensions given in the cell descriptor file, you will not need to specify these quantities here. At this stage, your *protodefs.g* might look something like

```
include compartments
include hhchan

EREST_ACT = -0.07
ENA      = 0.045
EK       = -0.082

include synchans
include protospike
```

Of course, your version will contain many illuminating comments!

After the relevant scripts have been included, we can create the library with the statements

```
create neutral /library
```

```

disable /library
pushe /library // Make these elements in the library

make_cylind_compartment /* makes "compartment" */

// Create prototype H-H channels "Na_squid_hh" and "K_squid_hh"
make_Na_squid_hh
make_K_squid_hh

// Make a prototype excitatory channel, "Ex_channel"
make_Ex_channel /* synchan with Ek = 0.045, tau1 = tau2 = 3 msec */

make_spike /* Make a spike generator element "spike"*/

pope // Return to the previous working element

```

You may notice that we have used a new GENESIS command, *disable*. We don't want to waste time having the elements in the library calculate anything, since they exist only to be copied into the elements that will actually be used in the simulation. When an element is disabled, the simulator will not attempt to update the fields of it or any of the child elements in its hierarchy. It may be re-enabled with the *enable* command.

The function *make_cylind_compartment* not only makes the element *compartment*, but it sets the default values of its fields and adds a new field *Shape*, which is initialized to "cylinder". As with *hhchan.g*, the script *synchans.g* uses global variables for the channel equilibrium potentials. The value for the glutamate excitatory channel *Ex_channel* is set with the statement "E_{Gl}u = 0.045". As this is the same as the value we used for the excitatory channel in Chapter 15, we can leave it as is. However, the two channel time constants, *tau1* and *tau2*, are "hard-coded" at 3 msec within the *make_Ex_channel* function. Fortunately, these are also the values that we want. If we were to require different values, we could set them to the correct values right after we create the channel. Finally, we need to create a spike generator, which will be linked to the soma by the cell reader. As the *make_spike* function in *protospike.g* creates exactly what we want (an element named *spike* with a unit amplitude and an absolute refractory period of 10 msec), we will use it here.

Before going on to construct the rest of the simulation, you may wish to test your *protodefs.g* file by executing it as a GENESIS script. If you have made no errors, it should execute without much happening on the screen. However, you may use the *le* and *showfield* commands to satisfy yourself that the proper prototype elements have been created in the library.

16.3 The Format of the Cell Descriptor File

Our cell will be assembled from the prototype elements in the library according to the specifications in the cell descriptor file. For our simulation we will use the example file *cell.p*, shown in Fig. 16.1. (This is often referred to as a *cell parameter file*, hence the required extension, “*.p*”.) We can best understand the format of this file by going through the file line by line.

```
// cell.p - Cell parameter file used in Tutorial #5

// Format of file :
// x,y,z,dia are in microns, all other units are SI
// In polar mode 'r' is in microns, theta and phi in degrees
// Control line options start with a '*'
// The format for each compartment parameter line is :
//name parent r theta phi d ch dens ...
//in polar mode, and in cartesian mode :
//name parent x y z d ch dens ...
// For channels, "dens" = max conductance per compartment unit area
// For spike elements, "dens" is the spike threshold
// Coordinate mode
*relative
*cartesian
*asymmetric

// Specifying constants
*set_compt_param    RM      0.33333
*set_compt_param    RA      0.3
*set_compt_param    CM      0.01
*set_compt_param    EREST_ACT   -0.07

// For soma, use the leakage potential (-0.07 + 0.0106) for EREST_ACT
*set_compt_param    ELEAK    -0.0594
soma none 30 0 0 30 Na_squid_hh 1200 K_squid_hh 360 spike 0.0

*set_compt_param    ELEAK    -0.07
dend soma 100 0 0 2 Ex_channel 0.795775
```

Figure 16.1 The cell descriptor file for the simple neuron model.

When cells are built “by hand,” the $[x, y, z]$ coordinate fields of the compartments are normally not used, except for graphical display of the cell geometry, or when the coordinates are needed to establish a pattern of connections in a network. However, the cell reader uses

these coordinates in order to determine the lengths of the compartments, and they are NOT optional. For an asymmetric compartment like that shown in Fig. 2.3, these coordinates are measured at the end that has the potential V_m . Using the option “*relative” allows one to specify these coordinates relative to the parent compartment without keeping track of the absolute location relative to the origin of the cell. As we will be using Cartesian coordinates, the option “*cartesian” is also specified. The “*asymmetric” option need not have been specified, as it is the default. It tells the cell reader that the various compartments will be constructed from copies of the asymmetric compartment in the library named *compartment*. If we had specified “*symmetric”, the compartments would have been constructed from the symmetric compartment prototype *symcompartment*.

Next, the values of the variables *RM*, *RA*, *CM*, and *EREST_ACT* are set, using the “*set_compt_param” option. These are internal variables that will be used by *readcell* for values of the specific membrane resistance and capacitance, specific axial resistance, and resting potential of the various compartments. The cell reader will use the values of *RM*, *RA* and *CM* to calculate and set the *Rm*, *Ra* and *Cm* fields from the compartment dimensions, ignoring any initial values set in the library prototype. Likewise, the *Em* and *initVm* fields are set from the *EREST_ACT* variable. However, we would like to set the soma *Em* voltage to the leakage potential rather than to the resting potential, as we have done in the previous tutorials. This is done by setting another internal variable *ELEAK* to -0.0594 V. After the soma is created, *ELEAK* is set to -0.07 , so that the dendrite *Em* will be given the proper resting potential. (Recall that the different value of *Em* was needed in the soma to compensate for the current flow through the Hodgkin–Huxley channels at the resting potential.)

A similar option, “*set_global” has the same effect on the internal compartment parameter variables as “*set_compt_param”, but also changes the value of the global script variables of the same name, which must have been previously declared. In addition, “*set_global” may be used to set the values of other previously declared global script variables that are not directly used by *readcell*. In general, it is best to use the option “*set_compt_param” in order to avoid the use of unnecessary global variables.

The line describing the soma compartment starts out by giving its name (“soma”) and its parent compartment (“none”). When reading the documentation for the cell reader, note that the terms “parent” and “children” are used somewhat differently than we have used them so far. If we have elements */cell/soma* and */cell/dend*, we would normally refer to the siblings *soma* and *dend* as children of the parent */cell*. When this same hierarchy of elements is created with the cell reader, the documentation refers to the dendrite as a child of the parent soma, because it is connected to the soma through the dendrite axial resistance. Thus, the line describing the dendrite (*dend*) lists *soma* as its parent, even though it will create */cell/dend*, not */cell/soma/dend*. To avoid confusion, we refer to *dend* as a *subelement* of */cell* whenever there is any ambiguity.

The soma that we created in the previous tutorials was $30\ \mu m$ long and $30\ \mu m$ in diameter. If we make our cell lie along the *x*-axis, that means that we should give (x, y, z, d) as $(30, 0,$

0, 30). After the coordinates and diameter of the compartment, we list the channels, giving the name of the prototype channel followed by the parameter “*dens*”. For a channel, this parameter is the maximum conductance per unit area of the compartment. For example, the function *make_Na_squid_hh*, defined in *hhchan.g*, sets the channel field *Gbar* to “1.2e3 * SOMA_A” siemens. In our previous simulation we were forced to use this “density” value of 1200, and had to explicitly override the default value of *SOMA_A* with our own value. The cell reader provides us with more flexibility, however. The density of 1200 is stated explicitly in the “*.p*” file, after the name of the channel, and the compartment dimensions are used with this parameter to set the value of *Gbar* when the channel is created from its prototype in the library. Thus, we need say nothing about *SOMA_A* in the *protodefs.g* file.

The spike generator is listed in the same manner as the channels. However, the *dens* parameter is used to give the spike threshold. The cell reader will then create the element */cell/soma/spike*, set its *thresh* field to 0.0, and add the INPUT message from the soma. The *abs_refract* and *output_amp* fields of the **spikegen** object are not set from the cell descriptor file. They will take on the values assigned to the prototype, unless these fields are explicitly set after the cell has been built. As in the previous chapter, we have used values that will produce a minimum interval (refractory period) of 10 msec between the spikes and give them a unit amplitude.

Next, we change the value of *ELEAK* to the appropriate value for the dendrite compartment. If we had wished to use different values for the specific resistances and capacitance, these parameters could be changed here. Then we give the name of the compartment (*dend*) its “parent” (*soma*), along with its coordinates, diameter and any of its subelements that need to be linked with messages. The geometrical parameters for the dendrite should be (100, 0, 0, 2) if it is to be 100 μm long and 2 μm in diameter. If we were using absolute coordinates instead of relative coordinates, we would have given the dendrite x-coordinate as 130.

In the previous tutorial, we gave the synaptically activated dendrite channel a maximum conductance of 5.0×10^{-10} siemen. Whether or not the *gmax* field of the prototype channel *Ex_channel* was set to this value, the cell reader will set the field of */cell/dend/Ex_channel* to a value calculated from the channel density parameter given here. You should verify that for the compartment dimensions used here, this results in a density of 0.795775 S/m^2 , as given in *cell.p*.

The GENESIS Reference Manual and the help file for the *readcell* command describe several other useful options. The “*compt” option is particularly useful for large models. A detailed cell model may use many compartments that are identical, except for their dimensions. These may often contain the same conductances, with the same conductance densities. In order to avoid the repetition of many identical long strings of channel specifications in the cell descriptor file, you may use the “*compt” option followed by the name of a compartment in the library. Rather than being a “naked” compartment like the */library/compartmet* element which we have used, this would typically consist of a compartment and its asso-

ciated subtree of channels and other subelements, linked with the appropriate messages. Once this option is specified, all following compartments will be copies of this element tree, with their channel conductances appropriately scaled to the dimensions of the compartment.

16.4 Modifying the Main Script to Use the Cell Reader

By making use of the two files *cell.p* and *protodefs.g*, we can eliminate much of the complexity of the *tutorial4.g* script. Copy this file into a new one with a different name and carry out the changes described below.

The cell parameters are either determined from the prototype definitions in *protodefs.g* or are read in from the parameter file, *cell.p*. Therefore all of the statements preceding the function definitions, except for those that set *tmax*, *dt* and *gmax* should be removed and replaced by the statement “*include protodefs*”. Likewise, the function definitions for *makecompartment*, *makechannel*, and *makeneuron* should be removed. The only function definitions remaining will then be those that involve the interaction of the cell with the outside world, the feedback connection and the graphics functions.

In the main script, the call to *makeneuron* should be removed and replaced by the statement

```
readcell cell.p /cell
```

Here, the two arguments specify the name of the cell descriptor file to be read and the path name of the resulting cell. As in the previous chapters, */cell* will be created from a **neutral** object.

At this point, you should have a script that is functionally equivalent to *tutorial4.g*. Try it out and verify that it works the same as *tutorial4.g*.

Using the cell reader to build this simple cell hasn’t saved us all that much work, as we still had to create all the prototype channels in the library and to provide the graphical interface. For a more complicated multi-compartmental cell, the effort saved could be considerable. This is because the same prototypes can be used in many compartments. In addition, the use of cell descriptor files provides a convenient mechanism for the exchange of cell models with other users of GENESIS.

16.5 The Neurokit Simulation

In this simulation, we still had to write a fair amount of GENESIS script language code in order to provide for the display of our results. The *Neurokit* simulation, which is found in the *GENESIS Scripts/neurokit* directory, takes us a step further by providing a graphical interface to the cell reader. This makes it possible to build complex multi-compartmental neurons and

to run elaborate single cell simulations without the necessity of writing GENESIS scripts. The *README* file that accompanies the simulation gives detailed instructions for using *Neurokit*. The next tutorial in this series shows how to implement this simulation within the *Neurokit* environment. In Chapter 19, we will learn how *Neurokit* and the cell reader are used with calcium concentration-dependent channels.

The subdirectory *Scripts/neurokit/prototypes* contains a large number of scripts, similar in format to *hhchan.g*, for creating a wide variety of channels. As with *hhchan.g*, these may be used independently of *Neurokit*. In order to encourage the exchange of new GENESIS simulation components, it is strongly recommended that you follow the conventions used in these scripts. The complete list of available channels may be found in the file *LIST* in this directory. In order to easily include these prototype scripts in your own simulations, include a statement like

```
setenv SIMPATH {getenv SIMPATH} /usr/genesis/Scripts/neurokit/prototypes
```

in either your *.simrc* file or your simulation scripts.

16.6 Exercises

1. In Chapter 7, we used *Neurokit* to experiment with a model of a burst firing molluscan neuron. The cell descriptor file *mollusc.p* in the *Scripts/burster* directory was used to create this cell. The file *ASTchan.tut.g* provides the functions needed to create the prototype channels and the file *userprefs.g* contains some statements that you can include in your *protodefs.g* file. (The statements at the end of *userprefs.g* are for use with *Neurokit*, and are described in the next chapter.) With these files and a modified version of *tutorial3.g*, duplicate the endogenous bursting behavior described in Sec. 7.5.2.
2. As a variation of the previous exercise, write a simple simulation to observe the soma membrane potential of the Traub model CA3 cell, with a current injection of 0.2 *nA* to the soma. Experiments with this model are also described in Chapter 7. The *Scripts/traub91* directory contains the files *CA3.p*, *traub91proto.g* and *userprefs.g*, which you may incorporate into your simulation. Your results should agree with those shown in the lower plot of Fig. 7.3.

Chapter 17

Building a Cell with Neurokit

DAVID BEEMAN

17.1 Introduction and Review

In the first four GENESIS programming tutorials, we covered the features of the GENESIS/XODUS script language needed to construct a simple model neuron. This neuron contains a dendrite compartment with a synaptically activated excitatory channel and a soma with Hodgkin–Huxley sodium and potassium channels. A source of randomly distributed spikes is used to excite the synapse. Action potentials produced in the soma trigger a spike generator that may be used to provide input to a synapse on another cell. In our model, we used a feedback connection to the cell’s own synapse that can be toggled on and off. Enough details of XODUS were introduced to create graphs for the membrane potential and channel conductance, along with buttons, toggles and dialog boxes for controlling the simulation.

The previous programming tutorial used the *readcell* function to build the same neuron from a data file. This is the preferred method for the construction of complex cells and the exchange of cell models with other GENESIS users. The tutorial also provided an introduction to the library of prototype cell components that are used with the *Neurokit* cell builder. In this tutorial, we will use the same neuron as an example for the use of *Neurokit* to construct a cell with a minimum of GENESIS programming. Although it is not necessary to have worked through the programming details of the previous five tutorials before beginning this one, you should read through them in order to understand the model that is being implemented here.

The *Neurokit* cell builder is a GENESIS simulation consisting of a main script *Neurokit.g* and several other included scripts. These make use of the cell reader to create a cell model, just as we did in the previous chapter. However, *Neurokit* also provides a graphical interface for controlling the simulation, modifying the cell model, stimulating the cell with various types of input, and displaying the value of fields in the various compartments.

In order to recreate our simulation with *Neurokit*, we need files for the cell reader similar to the ones that we used in the previous chapter. We will again use a cell descriptor file *cell.p*. We also need the scripts *compartments.g*, *hhchan.g*, *synchans.g* and *protospike.g* in the *neurokit/prototypes* directory. Again, these will be used to create the prototype elements from which the cell will be constructed. As before, we need a script that includes the above scripts and uses them to create the prototype elements under a **neutral** /library element. In the previous chapter, we chose to call this file *protodefs.g*. *Neurokit* requires that this file be called *userprefs.g*. It performs the functions of our *protodefs.g* file, plus a few others described in the following section.

Before beginning this tutorial, you should also familiarize yourself with *Neurokit* by running one of the *Neurokit*-based simulations described in Chapter 7. Alternatively, you might run the simulation in the *Neurokit* directory (*Scripts/neurokit*), using the *camit.p* cell parameter file which is found there. Try out some of the features that are described in the *README* file, which is available as the on-line help. If you wish to explore the channel editing capabilities of *Neurokit*, you may run the tutorial in *Scripts/channels*. Channel editing with *Neurokit* is also treated in a following chapter, in Sec. 19.2.2. Note that the *Neurokit edit channel* menu choice only refers to voltage-dependent ionic channels, and not to synaptically activated channels.

17.2 Customizing the *userprefs* File

When *Neurokit* is run, it begins by assigning default values to a number of global variables called *user-variables*. These variables are defined in the *Neurokit* directory file *defaults.g*, and are responsible for the initial values of most of the *Neurokit* dialog boxes, scales for graphs, and a good deal of what you see on the screen. Although you may wish to examine *defaults.g*, it should not be modified. Next, *Neurokit* looks for a file called *userprefs.g*. This file is responsible for creating the prototype elements in the library and for making any desired changes in the user-variables that were set in *defaults.g*. Unless you are running the example *camit* (CA mitral cell) demonstration simulation from the *Neurokit* directory, you will want to run *Neurokit* from another directory that contains your own customized *userprefs* file and your own cell parameter file. Typically, you will have a *.simrc* file that has set a path to the *Scripts/neurokit* and *Scripts/neurokit/prototypes* directories, so that this may be done. If the *userprefs* file is not found in the directory from which *Neurokit* is being run, the default *userprefs.g* in the *Neurokit* directory is used. This file, shown in Fig. 17.1,

may be used as a model for the construction of your own *userprefs* file. To illustrate this customization, we will go through the file, making the changes needed to construct our model neuron. If you are feeling impatient, you may consult the resulting *userprefs.g* listing in Appendix B. However, you will gain the most from this tutorial by developing the file piece by piece as you work through the tutorial.

17.2.1 Step 1

The first step is to include the scripts that define the functions to be used to create the prototype elements. In Chapter 19, we will create our own prototype channels. For now, we will build our neuron with channels that we can find in the prototypes directory. The *neurokit/prototypes/LIST* file provides a summary of these.

We need compartments (defined in *compartments.g*), sodium and potassium Hodgkin–Huxley channels (defined in *hhchan.g*), an excitatory synaptic channel (a **synchan** object), and a spike generator like the one we used in the previous tutorial. We can find the latter defined in *protospike.g*. In Fig. 17.1, the file *mitsyn.g* defines functions for creating both an excitatory channel (*make_glu_mit_upi*) and an inhibitory channel (*make_GABA_mit_upi*). Note that, unlike the names in our “generic” *synchans.g* file, the prototype channel names follow the convention recommended in the *neurokit/prototypes/README* file. For example, *glu_mit_upi* is a glutamate-activated mitral cell channel, authored by Upi Bhalla. In general, you should look for a file that defines a prototype closest to the one you need. In some cases, it may be necessary to modify an existing file to produce the desired prototype, or to change the default field values, once the prototype is created. For our model, we can use the file *synchans.g*, which has exactly what we need. Thus, the first section of our *userprefs* file will be fairly similar to that of the default version and to the statements given in Sec. 16.2.2. It will contain the statements:

```
include compartments /* file for standard compartments */
include hhchan    /* file for Hodgkin--Huxley Squid Na and K channels */
include synchans   /* file for synaptic channels */
include protospike /* file which makes a spike generator */
```

17.2.2 Step 2

Neurokit begins by creating a neutral element */library* to contain the prototypes, so our second step should be to change to this element and execute the functions that will create the desired prototype elements. However, we should first consider what we need to do in order to properly set the values of the internal fields of these prototypes. In the case of compartments, all the relevant fields can be set from information contained in the cell parameter file. For channels, the maximum channel conductance is the only field that is determined from the cell parameter file. *Neurokit* allows some other fields to be set from dialog boxes, but many

```

// genesis - default neurokit/userprefs.g file
echo Using default user preferences!

// Step 1 - Including script files for prototype functions
/* file for standard compartments */
include compartments
/* file for Hodgkin--Huxley Squid Na and K channels */
include hhchan
/* file for Upi's mitral cell channels */
include mitchan
/* file for Upi's mitral cell synaptic channels */
include mitsyn

// Step 2 - Invoking functions to make prototypes in the /library element
pushe /library // make all subsequent elements in the library

/* Make the standard types of compartments */
make_cylind_compartment      /* makes "compartment" */
make_sphere_compartment      /* makes "compartment_sphere" */
make_cylind_symcompartment   /* makes "symcompartment" */
make_sphere_symcompartment   /* makes "symcompartment_sphere" */

/* These are some standard channels used in .p files */
make_Na_squid_hh            /* makes "Na_squid_hh" */
make_K_squid_hh              /* makes "K_squid_hh" */
make_Na_mit_hh                /* makes "Na_mit_hh" */
make_K_mit_hh                /* makes "K_mit_hh" */

/* There are some synaptic channels for the mitral cell */
make_glu_mit_upi             /* makes "glu_mit_upi" */
make_GABA_mit_upi             /* makes "GABA_mit_upi" */

pope    /* returning to the root element */

// Step 3 - Setting preferences for user-variables.
user_syntype1 = "glu_mit_upi"
user_syntype2 = "GABA_mit_upi"

```

Figure 17.1 The *userprefs.g* file for the default *Neurokit* simulation. For brevity, some of the comments have been removed.

would have to be changed, using the GENESIS *setfield* command, if the desired values are not set when the prototypes are created. Thus, we should examine the channel creation functions in order to see what changes we might need to make within the *userprefs* file.

Looking at the listing for *hhchan.g* in Appendix B, we see that it makes use of the global variables *EREST_ACT*, *ENA*, and *EK* to define the resting potential and the Na and K equilibrium potentials. The functions *make_Na_squid_hh* and *make_K_squid_hh* use these values when creating the channels. If they are not the values we want, we should modify them within *userprefs.g*. This should be done at a point *after* the script is included, but *before* the functions are invoked. As the default values had been modified from the original Hodgkin–Huxley squid values for use in a mitral cell simulation, we will need to change them to the proper values, as we did in Chapters 14 through 16. (You may note that *hhchan.g* assigns a value to a fourth global variable, *SOMA_A*. Why may we leave this as is?)

As in the *protodefs.g* file from the previous tutorial, we can use the *make_Ex_channel* function to create the glutamate-activated channel *Ex_channel* with all the proper default field values. Although it isn't used by the cell model that we are going to create, we might as well put a GABA channel *Inh_channel* in the library, in case we want to edit the cell to include it later. Chapter 15 describes how a spike generator linked to the soma may be used to translate somatic action potentials to presynaptic neurotransmitter release. As before, we will use the *make_spike* function in *protospike.g* to create an element named *spike* with a unit amplitude and an absolute refractory period of 10 msec. Thus, the second section of *userprefs.g* contains:

```

pushe /library
make_cylind_compartment           /* makes "compartment" */

/* Assign some constants to override those used in hhchan.g */
EREST_ACT = -0.07      // resting membrane potential (volts)
ENA     = 0.045        // sodium equilibrium potential
EK      = -0.082        // potassium equilibrium potential

make_Na_squid_hh    /* makes "Na_squid_hh" */
make_K_squid_hh    /* makes "K_squid_hh" */

make_Ex_channel    /* synchan with Ek = 0.045, tau1 = tau2 = 3 msec */
make_Inh_channe   /* synchan: Ek = -0.082, tau1 = tau2 = 20 msec */

make_spike         /* Make a spike generator element */

pope              /* return to the root element */

```

Of course, one does not have to rely on the functions that are defined in the *neurokit/prototypes* files. *Neurokit* sets the SIMPATH to include “*./prototypes*”, so you may

have your own prototypes directory with scripts containing your own function definitions.

17.2.3 Step 3

The final step is to make any needed changes in the user-variables from the values given in *defaults.g*. Although most of these may be set from dialog boxes within *Neurokit*, it is far more convenient to have *Neurokit* start up configured for the simulation that you want to run. In this part of the tutorial, we will set some of the more common user-variables.

The *Neurokit* file menu contains default values for the name of the cell to be created and the source file name (the cell descriptor file) that will be loaded when you click on Load from file. In Chapter 16, we created a cell called */cell* from the file *cell.p*, so we will start with:

```
user_cell = "/cell"  
user_pfile = "cell.p"
```

The run cell selection brings up a SIMULATION CONTROL PANEL form with dialog boxes for many parameters that we will want to customize for our simulation. Figure 17.2 shows the display that is produced with the *camit* simulation. It would be best to run *Neurokit* as you read through this tutorial. If you do this with the *userprefs* file you have created so far, you will be able to experiment by changing the dialog boxes by hand as you edit your *userprefs* file to change the user-variables. The parameters that we are most likely to want to change are the run time, the simulation time step (*clock*), and the number of steps between update of the graphs (*refresh_factor*). In the previous tutorials, we ran the simulation for 100 msec, using a time step of 0.05 msec. We can save some execution time by plotting the graphs every five simulation steps, instead of plotting at every simulation step. These values may be set with the statements:

```
user_runtime = 0.1  
user_dt = 50e-6 // 0.05 msec  
user_refresh = 5
```

We will also need to modify some variables that appear in the dialog boxes under the ELECTROPHYSIOLOGY heading of the SIMULATION CONTROL PANEL. Some of these dialog boxes appear and disappear according to the type of stimulation that is applied to the cell. Table 17.1 lists the types of stimulation, the dialog box labels, and the associated user-variables.

We will experiment with the values of most of these later on in the tutorial. You may assign reasonable initial values to these in the *userprefs* file now, or may do it later after having gained some experience with the simulation. In the earlier tutorials, we found that 0.3 nA was a good value to use for the injection current, so you may set *user_inject* to

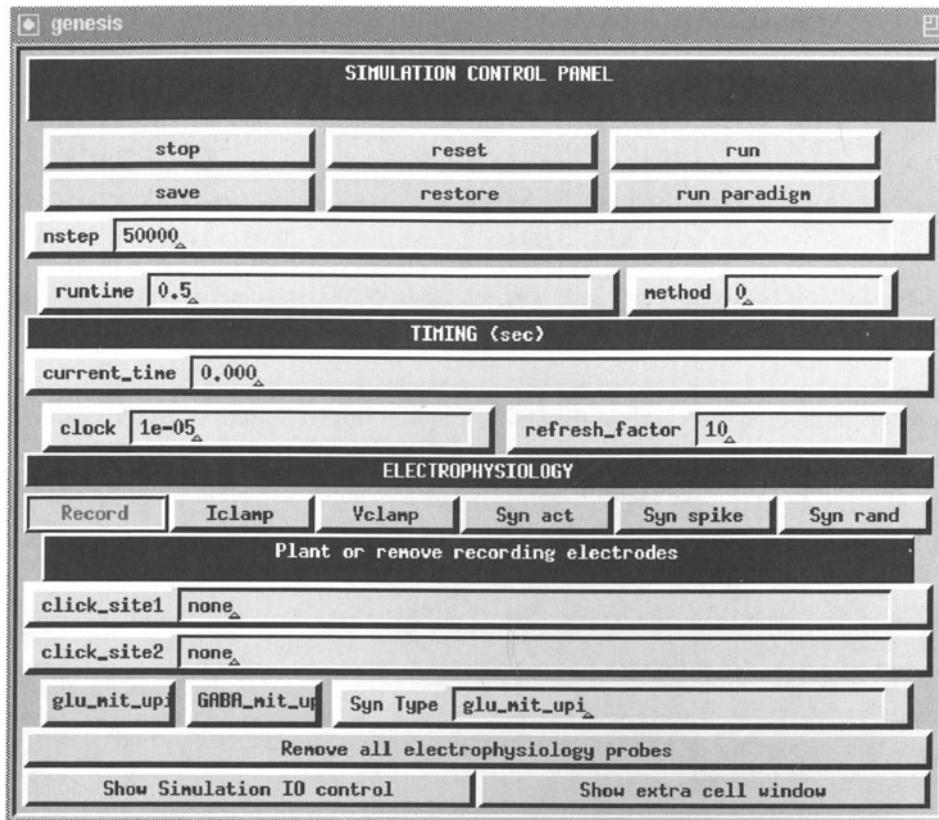


Figure 17.2 The *Neurokit* SIMULATION CONTROL PANEL form.

this value now. The two boxes labeled `click_site1` and `click_site2` obtain their values when you click on a compartment in one of the two Cell Windows, for either recording or stimulation. Finally, there is a dialog box labeled `Syn Type`. This contains the name of the channel that will be used for one of the three forms of synaptic stimulation. The contents of this box may be set by hand, or by clicking on one of the two buttons to the left. We can set the labels of these buttons to the two available synaptically activated channels by including the statements

```
user_syntype1 = "Ex_channel"
user_syntype2 = "Inh_channel"
```

in the `userprefs` file, as was done in the default `userprefs` file for the *camit* simulation.

There are also one or two Cell Windows that each contain a diagram of the cell and a graph with axes scaled appropriately for the quantity to be plotted. The simulation that we

<i>Stimulation</i>	<i>Dialog Box Label</i>	<i>User-variable</i>
Iclamp	Inject (nA)	user_inject
Vclamp	Clamp voltage (mV)	user_clamp
Syn act	Amount of synaptic activation	user_activ
Syn spike	Weight of spike	user_spike
Syn rand	Spike rate (Hz)	user_rate
	Spike weight	user_weight

Table 17.1 The dialog boxes and user-variables associated with the various types of stimulation that may be applied to the cell.

are trying to recreate had a graph to plot the soma membrane potential, and a second graph to plot the conductance of the excitatory channel. We can specify the axis scaling for these two graphs to be the same as we used before, if we add the statements:

```
user_ymin1 = -0.1
user_ymax1 = 0.05
user_xmax1 = 0.1
user_xmax2 = 0.1
user_ymin2 = 0.0
user_ymax2 = 5e-9
```

We also need to use the *user_numxouts* variable to specify that two graphs will be shown, and we need to specify what will be plotted in each graph. The *defaults.g* file specifies values of “Vm” and “.” for the user-variables *user_field1* and *user_path1*. This means that the *Vm* field will be plotted for the compartment that is selected for recording in the first Cell Window. This is just what we want for the first graph. However, these same defaults are also specified for the second graph. For our second Cell Window, we would like to be able to plant a recording electrode in the dendrite compartment and plot the conductance *Gk* of the excitatory channel *Ex_channel*. We can make these changes with the statements:

```
user_numxouts = 2
user_field2      = "Gk"
user_path2 = "Ex_channel"
```

There are also *scale* buttons that will bring up menus to change the representation of the cell which is shown in each window. The *Neurokit README* and *defaults.g* files provide information on the variables that appear in these menus. Most of these correspond to fields of the **xcell** widget, as described in the GENESIS Reference Manual. For now, we will use the default values.

17.3 The Cell Descriptor File

The format of the cell descriptor file was discussed in Chapter 16, so you may wish to review that section of the tutorial. For our simulation under *Neurokit*, we can use the same *cell.p* file.

You should recall that the cell reader uses the “*set_compt_param” option to specify its own internal variables *CM*, *RM* and *RA* in the cell descriptor file in order to calculate the fields *Cm*, *Rm* and *Ra* from the compartment dimensions, ignoring any initial values set in the prototypes. The *Em* and *initVm* fields are similarly set from the *EREST_ACT* variable. The *ELEAK* variable may be used if it is necessary to set a different leakage potential for *Em*. Also, the maximum conductance for a channel is set from the *dens* parameter and the dimensions of the parent compartment, overriding any value set in the prototype. However, all other channel parameters remain the same as those in the prototype library. Fortunately, *Neurokit* provides dialog boxes for changing many of them.

17.4 Some Experiments Using Neurokit

Now that you have a customized *userprefs.g* file and a *cell.p* file, you are ready to try some experiments with *Neurokit*. You should run GENESIS from the directory that contains these files, and then type “*Neurokit*” to the GENESIS prompt. If all is well, the *Neurokit* title bar should eventually appear at the top of the screen. If GENESIS cannot find *Neurokit*, make sure that your *.simrc* file has properly set the SIMPATH. If you get fatal errors during the loading of *userprefs.g*, look for syntax or typographical errors in this file. Once *Neurokit* is running, click on the *file* option on the title bar. (As usual, when we say “click on . . .” we mean “click the left mouse button on . . .”, unless another button is specified.) Next, click the *Load from file* button on the *file* menu, and then click *run cell* on the title bar. You should see the **SIMULATION CONTROL PANEL** with the timing parameters you provided and two Cell Windows with properly scaled graphs.

Initially the **ELECTROPHYSIOLOGY** buttons will show that you are ready to plant recording electrodes. Click on the soma in the first Cell Window and on the dendrite in the second Cell Window. The two *click_site* dialog boxes should now show “/cell/soma” and “/cell/dend”.

As a start, let’s do a simple current injection experiment to make sure that the soma compartment is working properly. After clicking on *Iclamp*, you should see a dialog box showing that the injection current is 0.3 nA. Click on the soma in the first Cell Window to plant an injection electrode. Now click on *run* in the **SIMULATION CONTROL PANEL**. After you have satisfied yourself that this provides the expected action potentials, remove the injection electrode by clicking on the soma with the middle mouse button and click on *reset* to clear the graph.

We are now ready to try applying synaptic activation. We will start by using mouse clicks to deliver spikes to the *Ex_channel* channel. This channel name should be displayed in the Syn Type dialog box. If not, the name can be entered in the box “by hand,” or by clicking on the button labeled “*Ex_channel*” at the left.

First, click on the Syn spike button under the ELECTROPHYSIOLOGY heading. Note that the MOUSE display at the top of the two Cell Windows now shows that the left button is labeled “SynSpike” and that the middle button is labeled “UnSpike”. Unless you have changed the *user_spike* variable in the *userprefs* file, the Weight of spike dialog will show the default value of 1.0. The maximum conductance (*gmax*) of a synaptically activated channel is reached when a spike with a unit amplitude is delivered to the channel, so this is a reasonable amplitude to use.

Now, click on run in the SIMULATION CONTROL PANEL. After the membrane potential has leveled off to about -70 mV , click on the dendrite compartment in the second (right) Cell Window. You should see a nearly linear rise of the channel conductance, followed by an exponential decay with a time constant of about 3 msec . Try clicking several times rapidly in succession, in order to let the conductance and postsynaptic potential build up to a level sufficient to create an action potential in the soma. In order to check the consistency of these results, we can inspect the channel parameters by selecting edit cell from the top menu bar.

Click on the dendrite compartment in the Cell Window (lower left) and then on the *Ex_channel* channel in the Compartment Window (upper right). The Parameter Window should display the dendrite dimensions and the maximum channel conductance in S/m^2 . Of course, *gmax* can also be found by typing “showfield /cell/dend/*Ex_channel* gmax” to the GENESIS prompt. Verify that this agrees with the maximum conductance provided by a single spike.

In order to deliver spikes at random intervals, select run cell again and click on Syn rand in the ELECTROPHYSIOLOGY menu. Dialog boxes will appear for both Spike rate and Spike weight. Try a spike rate of 200 Hz , and leave the weight at 1.0, as before. The left and middle mouse buttons will now be labeled “RandSyn” and “UnRand”. Click the left button on the dendrite compartment in Cell Window 2, and then run the simulation. Repeated runs (after reset) will continue to deliver random spikes until you click the middle button in the Cell Window 2 dendrite compartment. (Note that switching to another form of input to the neuron, such as Iclamp or Syn spike, will not remove the random spike input unless you first use the middle button to remove the input.)

The Syn act button delivers a constant synaptic input. Using the default value of 1000 for the Amount of synaptic activation, run the simulation and click the left button in the Cell Window 2 dendrite compartment. Note that the channel conductance levels off at about $4.0 \times 10^{-9}\text{ siemen}$. Can you explain why? Be sure to use the middle button to remove the activation when you are through.

Now, how about the feedback connection that we used in Chapters 15 and 16? *Neurokit*

was designed for single cell modeling, without connections between the cells, so we will have to enter some additional GENESIS commands in order to make the connection. As we want to make a synaptic connection between the spike generator and the *Ex_channel* channel, we enter a command at the GENESIS prompt to send a SPIKE message to the channel:

```
addmsg /cell/soma/spike /cell/dend/Ex_channel SPIKE
```

We also need to assign a synaptic weight and a delay for propagation along the axon. In the previous tutorials, we used a weight of 10 synaptic connections and a delay of 5 msec. We also need to set some parameters for the spike generator. The cell parameter file only lets you set the threshhold for the spike generator. We also need an amplitude and an absolute refractory period. As with our previous implementations of this cell model, we want to give each spike a unit amplitude and to give the spikes an absolute refractory period of 10 msec. Thus, we need to give the additional commands:

```
setfield /cell/dend/Ex_channel synapse[0].weight 10.0 \
          synapse[0].delay 0.005
setfield /cell/soma/spike output_amp 1 abs_refract 0.010
```

By using *showfield* to find the number of synaptic connections to *Ex_channel*, you should verify that the index “0” is the proper one to designate this synaptic connection. After entering these commands, perform the *Syn rand* experiment again. You should notice that after the first action potential occurs, the feedback produces a continous stream of action potentials, as in Chapters 15 and 16. As we did in these tutorials, we can delete the connection with the command

```
deletemsg /cell/soma/spike 0 -out
```

So far, we have done nothing with the inhibitory synaptic channel that is lying dormant in the library. Use *edit cell* to paste in the *Inh_channel* channel. Use the cell parameters form to set the maximum conductance to the same as that of the glutamate channel. Select this as the *Syn Type* and inject some current to the soma in order to get the cell firing. We would like to deliver some activation to this channel to prevent the cell from firing. However, we would also like to see a plot of the conductance from this channel instead of that from the glutamate channel. This can be changed with the *scale* button at the top of Cell Window 2. Change the *fieldpath* dialog entry to “*Inh_channel*”. Now use either *Syn spike* or *Syn rand* to inhibit the firing that is produced by the current injection.

17.5 Exercises and Projects

1. In the simulations of Chapters 15 and 16, we had a toggle button to allow us to add and delete the feedback connection between the axon and the excitatory synaptic channel. Here, we had to resort to the rather awkward expedient of entering long GENESIS commands. Will *Neurokit* allow us to extend its capabilities and build in the feedback connection along with a toggle to take it in and out? Can we have dialog boxes to enter the value of the weight and delay parameters for the synaptic connection? Fortunately, the answer is “yes.” The `run paradigm` button in the **SIMULATION CONTROL PANEL** will invoke a user-supplied function called `do_paradigm`. The definition of this function and the statements needed to create a form with the needed dialog boxes and toggle can reside in a supplementary script that is brought into the simulation with the `include` statement. This may be done in your `userprefs` file or may be done at the GENESIS prompt after your simulation has been started. The demonstration simulation in `Scripts/vclamp` provides an example of such a paradigm file. For another example, take a look at the script `I_plots.g` from the *Burster* tutorial of Chapter 7. Use this information to write a function that will cause the `run paradigm` button to toggle the feedback connection on and off.
2. There are a number of experiments that one can do with just a “naked” soma in order to study the effects of various types of voltage-activated ionic channels. Modify your `cell.p` file to create just this simple soma with the two Hodgkin–Huxley channels. Also modify your `userprefs` file to put a non-inactivating muscarinic potassium current (the “M-current”) in the library. The conductance may be created with the function `make_KM.bsg_yka` in the `neurokit/prototypes` file `yamadachan.g`. This file contains functions to create several conductances that were used in a model of a bullfrog sympathetic ganglion neuron by Yamada, Koch and Adams (1989).

Start by calculating and making a plot of the current-frequency (f-I) relationship of a standard Hodgkin–Huxley patch of membrane (i.e., containing only the Na and K channels). That is, inject a constant amount of current for, e.g., 200 msec, and plot the resulting frequency of spikes versus the amplitude of the injected current. Reduce the potassium conductance and describe how the f-I curve changes. Next, use `edit cell` to paste in the M-current channel and recompute the f-I curve. How does it change? Note that even if very large input currents are applied, the spiking frequency will not become arbitrarily large. What underlying parameter limits the spiking frequency?
3. In Chapter 7 (Sec. 7.3.2), we briefly mentioned a low threshold calcium current that contributes to burst production in thalamic relay cells. The file `THALMODES.g` in the `neurokit/prototypes` directory implements the channel models that are described by

McCormick et al. (1992). Use these channels to create a single compartment bursting thalamic relay cell model.

4. Try to recreate the results obtained by Connor and Stevens (1971c) for their model of *Anisidoris* gastropod neurons. The channel prototypes are contained in *CSchan.g*.
5. If you have worked through the programming of the tutorial in Chapter 15, you may be interested in doing some “snooping” beneath the surface of *Neurokit*. While the simulation of our simple cell is running with `Syn ran` selected for synaptic input to the `Ex_channel` channel, use the `showfield` and `showmsg` commands to discover how the input is provided to this channel. How does this compare with the way that the random input was implemented in Chapter 15? What connections or messages are used to provide the `Syn act` option?

Chapter 18

Constructing Neural Circuits and Networks

MICHAEL VANIER AND DAVID BEEMAN

18.1 Introduction

In this chapter, we demonstrate how to use GENESIS to set up a simple network of biologically realistic neurons. This will not be a “neural network” in the usual sense of a network of highly abstract units with no direct connection to biological neurons (such as a backpropagation network). Rather, the approach we take is to simulate a group of biological neurons at a moderate level of detail and then connect them in a network. In the process we discuss a number of GENESIS functions that are used for this purpose, as well as a few script commands that have not been described earlier in this book. Our examples are taken from a tutorial simulation called *Orient.tut*, which is a simplified model of orientation selectivity originally written by Upinder S. Bhalla. This tutorial contains several script files, of which about half deal with setting up the XODUS graphical user interface. We do not discuss these scripts in this chapter; the GENESIS commands used to set up the interface are described in Chapter 22, “Advanced XODUS Techniques.” The emphasis in this chapter is on showing you how to use GENESIS commands whose primary purpose relates to simulation of networks of neurons. Commands that have been discussed in detail in other chapters are mentioned only briefly here. Another example of a large network simulation in GENESIS is provided in Chapter 9, on the piriform cortex simulation.

18.2 The *Orient_tut* Simulation

The *Orient_tut* simulation is a simplified model of orientation selectivity in the visual system. It is available in the *Scripts/orient_tut* directory in the GENESIS distribution. This simulation consists of two groups of cells, “retinal” cells and “V1” cortical cells. The retinal cells generate spikes randomly at a rate controlled by the simulation. They in turn make synaptic contacts with two types of V1 cells: horizontal bar detectors and vertical bar detectors. The simulation allows you to sweep a horizontal or vertical bar pattern across the “retina.” You will observe that sweeping a horizontal bar across the retina causes the horizontally selective cells in V1 to become activated, whereas sweeping a vertical bar across the retina causes the vertically selective cells in V1 to become activated. As you will see, this selectivity results from the specific patterns of connectivity between the retinal cells and the horizontally and vertically selective V1 cells. The degree of orientation selectivity is quite weak: occasionally a vertical bar will cause a horizontally-selective cell to become active and vice versa. The exercises at the end of the chapter discuss ways of improving the orientation selectivity. One should bear in mind that this is not a very realistic simulation of the mammalian visual system; for instance, there is no representation of the lateral geniculate nucleus (LGN) and there are no feedback connections within V1. The aim of this simulation is not to provide a state-of-the-art model of orientation selectivity but to show in a relatively simple setting most of the commands that are used in setting up network simulations in GENESIS.

18.3 Running the Simulation

The simulation is started by changing to the *Scripts/orient_tut* directory and typing “`genesis Orient_tut`”. A graphical display will come up like that in Fig. 18.1. The control panel is in the upper left corner. This contains various buttons, toggles and dialog boxes, all of whose functions are explained in the *README* file for the simulation. For the purposes of this chapter, the most important buttons are the two buttons on the top row, called `sweep_vert` and `sweep_horiz`. Clicking the mouse on `sweep_vert` causes a vertical bar of retinal cells to become active, slowly creeping from left to right, accompanied by some background firing. You can look at the firing pattern of the retinal cells in the display to the right of the control panel, under the label `Input Pattern`. You will see squares that flick from the background color to red and back again; this happens whenever a spike occurs in the corresponding cell. There are 100 such cells in the display, and they are arranged according to their *x-y* coordinates. This part of the display is actually composed of two XODUS objects called `xview` and `xdraw`; see Chapter 22 for details. We refer to such a display henceforth as a *draw widget*.

You can stop the simulation by clicking on the `stop` button. If you click on the

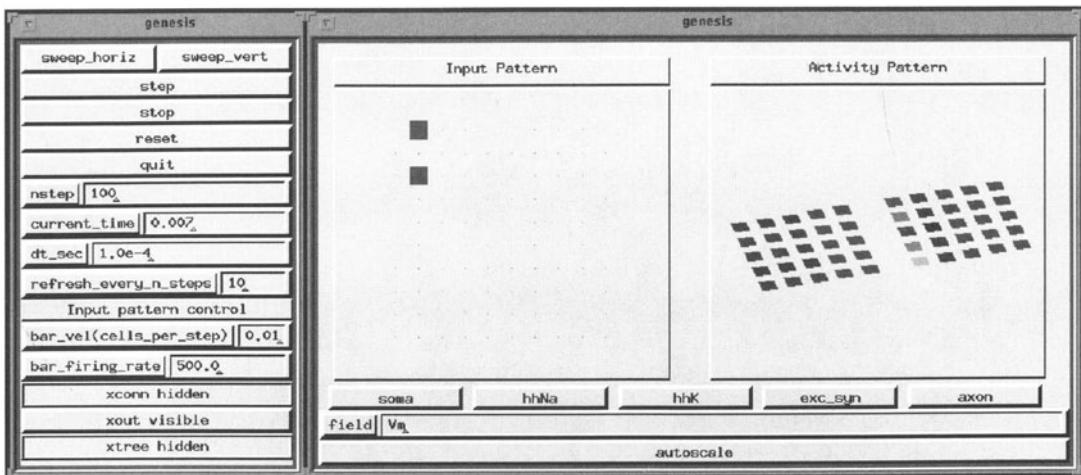


Figure 18.1 The upper portion of the *Orient_tut* simulation display. The window to the right of the control panel shows the retinal receptor cell array and the two planar arrays of horizontally and vertically selective V1 cells.

sweep_horiz button, a bar of active cells will creep from the bottom of the draw widget to the top. The outputs of the cells are displayed in a different draw widget, under the label Activity Pattern. This widget has two groups of 25 squares, representing the 25 horizontally and vertically selective V1 cells (horizontal to the left, vertical to the right). You can view the value of any of several fields in this widget, but the most useful one (and the default) is the membrane potential (V_m) of the V1 cells. The V_m field of the cell's soma compartment is displayed as the color of the square, with blue being hyperpolarized and red being depolarized. When a horizontal bar is swept across the retina, observe how a bar of horizontal V1 cells becomes active and tracks the motion of the bar across the retina. There will also be some activity in the vertically selective cells, but considerably less. Conversely, when a vertical bar is swept across the retina, a bar of vertically selective cells becomes active and tracks the inputs, whereas the horizontally selective cells are much less active. This result follows from the different connectivity patterns of the horizontally and vertically selective cells. This can be viewed by clicking on the toggle marked xconn hidden. This brings up yet another draw widget with three cell displays shown in perspective; the leftmost one represents the retinal cells, the upper right one represents the vertical cells and the lower right one represents the horizontal cells. Clicking the mouse on a retinal cell will show you the projection pattern of that cell in both V1 fields, whereas clicking on a V1 cell will display the receptive field of that cell in the retinal field. You should be able to figure out what confers the orientation selectivity upon the V1 layers by looking at the projective and receptive fields in this way.

For more details on the graphical interface, see Chapter 22 or the *README* file accom-

panying the simulation. Now, we will discuss the GENESIS commands needed to set up the network for this simulation.

18.4 Creating a Network Simulation

The process of setting up a neural network simulation in GENESIS can be divided up conceptually into several stages. First, the basic low-level components of the simulation that are used to construct cells must be specified. These include compartments, ion channels, and other elements that may include elements to simulate calcium fluxes, detect when spike thresholds have been reached, and so on. The second stage is to link these low-level elements into single cells representing the different cell classes to be incorporated into the model. The third stage is to create arrays of these cells corresponding to the biological structures being modeled. In our case, this means creating an array of retinal cells, and arrays of both vertically and horizontally selective V1 cells. In the fourth stage, we connect the cells in a network, specifying the connection strengths, axonal and synaptic transmission delays, and all other parameters of the synaptic connections. Then, we create any needed external inputs to the system. The final stage is to set up the user interface and output routines that allow the user to run the simulation, view the data being generated, and/or save the data to disk for further analysis.

Of course, you may not always rigorously follow this order. Usually, you will want to do some testing and debugging while you are constructing your network. This may mean providing some input to the system and some graphical display before you have completely finished stage four. Section 18.7.3 describes some utility functions that will be useful in debugging network simulations.

The focus of this chapter is on the third and fourth stages (setting up the network) since other chapters give more details on setting up single cell models and user interfaces. However, we will first briefly describe the manner in which the *Orient_tut* script files handle the first two stages.

18.5 Defining Prototypes

In the *Orient_tut* simulation, the prototype elements are defined in the files *constants.g* and *protodefs.g*. The first script defines the values of constants such as the Nernst equilibrium potentials for the different ion classes (Na, K and Cl), resting potentials, single channel conductances, channel densities, cell dimensions, axonal and synaptic propagation delays, and so on. In some GENESIS simulations the Nernst potentials are computed by the *nernst* object as the simulation is running; however, since the *Orient_tut* simulation is somewhat less detailed, the Nernst potentials are assumed to be constant. The *protodefs.g* script defines the component elements needed to construct the cells. The different objects used include

a basic compartment object, active Na^+ and K^+ Hodgkin–Huxley type ionic channels, synaptic channels (depolarizing Na^+ channels, hyperpolarizing K^+ channels and shunting Cl^- channels), and a spike detector **spikegen** object. These objects have been described in previous chapters, especially Chapter 15.

In *protodefs.g* we create the **neutral** element */library* to store the basic synaptic and ionic channel types, as well as a basic compartment element and a spike detector (**spikegen** element). Then we disable the library using the command “**disable /library**” so that the elements in */library* are not simulated during the simulation. The purpose of this is to use the library elements as templates that can be copied to other places where they will be linked into the main simulation, as described in Chapters 16 and 17. In this case they will be used to make prototype cells, which we will copy into the network arrays. Another approach would be to make the library prototypes into GENESIS extended objects, as discussed in Sec. 14.5. However, the approach described here is adequate for our purposes.

After *protodefs.g* has been loaded, all the prototype channels and compartments are subelements of the */library* element. Now we want to use these elements to build up prototype cells for the simulation. There are two types of cells: the receptor cells, which can be thought of as very crude representations of retinal ganglion cells, and the V1 cells, which represent the synaptic “targets” of the retinal cells. These cell types are defined in the script files *retina.g* and *V1.g*, which we describe next.

In *retina.g*, a sample “receptor” cell is created in the library. This is not a real cell in the sense of having compartments or ion channels. It is merely a spike generator that can be connected to postsynaptic cells in the same way as a cell. Only one object is required for this: the **randomspike** object. **randomspike** is a spike generator that generates random (Poisson-distributed) spikes at an average rate set by its *rate* field; for more details, see Chapter 15.

The *retina.g* script creates a prototype receptor cell called */library/rec*, where *library* and *rec* are both **neutral** elements. It then creates the **randomspike** element (random spike generator) called *input* in *library/rec* and sets its average firing rate and absolute refractory period.

The script *V1.g* works in a similar manner, building a V1 cell in the library (as */library/soma*). In this case, the base element (*/library/soma*) is derived from a **compartment** object, not a **neutral** object. We could have made the base element from a **neutral** object, but conceptually all the other parts of the V1 cell are connected to the soma anyway, so this was more convenient. The V1 cell consists of a single compartment (*/library/soma*) that is linked to Hodgkin–Huxley Na^+ and K^+ channels, excitatory and inhibitory synaptically activated channels, and a **spikegen** element, which detects spikes occurring in the cell.

18.6 Creating Arrays of Cells

Now that we have a prototype receptor cell and a prototype V1 cell, we are ready to create the retina and V1 arrays. The statement

```
createmap /library/rec /retina/recplane \
{REC_NX} {REC_NY} \
-delta {REC_SEPX} {REC_SEPY} \
-origin {-REC_NX * REC_SEPX / 2} {-REC_NY * REC_SEPY / 2}
```

in *retina.g* is used to make multiple copies of *rec* and its children into */retina/recplane*, arranging their *x*-*y* coordinates on a two-dimensional grid. */retina/recplane* is a **neutral** element created by the command for the purpose of storing the copies of the receptor cell. The usage for the *createmap* command is

```
createmap source dest Nx Ny -delta dx dy -origin xmin ymin -object
```

where *Nx* and *Ny* are the number of cells on a side in the *x* and *y* directions (giving a total of $Nx \times Ny$ cells in all), *dx* and *dy* are the physical separations in the *x* and *y* directions, and *xmin* and *ymin* give the position of the first element to be created (i.e., with the lowest *x*-*y* values). If the entity to be copied is a GENESIS object in its own right (as opposed to the base of a tree of elements) you should use the *-object* option. This is mainly for use with GENESIS extended objects (see the GENESIS Reference Manual) and is not needed in this simulation. Here *REC_NX* and *REC_NY* (the dimensions of the receptor cell array, i.e., the number of cells on a side in the *x* and *y* directions) are both 10, and the cell separations (*REC_SEPX* and *REC_SEPY*) are each 40×10^{-6} (meters; i.e., $40 \mu m$ — all units are SI unless otherwise noted).

Thus, the command “le */retina/recplane*” gives “*rec[0-99]/*”, meaning that it has created *rec[0]* through *rec[99]*. The “/” indicates that each element has child elements. In this case the only child element will be the **randomspike** element *input*. The “cells” *rec[0]* through *rec[99]* are **neutral** elements because all they need to do is to contain the random spike generator. It would also have been possible to create these copies with the command “copy /library/rec /retina/recplane/rec[0] -repeat 100”. However, the use of *createmap* assigns values to the *x* and *y* coordinate fields of the elements corresponding to their locations on the grid. These coordinates will be used not only to display the cells in a draw widget, but to assign synaptic connections, synaptic weights and propagation delays.

Occasionally, it might also be useful to specify a non-rectangular geometry for the positions of the retinal cells. For example, we might wish for the retinal cells to be arranged so that they fill up the interior of a circular region centered on the origin with a radius of 200 microns. In this case *createmap* will not work, since it arranges the cells in a rectangular grid. However, we could write a script function to implement this arrangement as follows:

```

function make_circular_retina
    int i,j
    int k = 0
    for(i = -5; i <= 5; i = i + 1)
        for(j = -5; j <= 5; j = j + 1)
            if({i*i + j*j <= 25})
                copy /library/rec /retina/recplane/rec[{k}]
                position /retina/recplane/rec[{k}] \
                    {20e-6 * i} {20e-6 * j} 0.0
                k = k + 1
            end
        end
    end
end

```

This function is instructive in that it demonstrates the use of some of GENESIS' looping and conditional constructs. Like any modern programming language, GENESIS has a full range of such commands, including *for*, *foreach*, *while*, *if/else*, and so on. All such commands terminate with an *end* statement. The syntax of these commands is similar to that of the corresponding C functions (or C shell, in the case of *foreach*); for full details, see the GENESIS Reference Manual. In this case the function generates pairs of integers ranging from -5 to 5 and tests whether the sum of their squares is less than or equal to 25 , i.e., whether the numbers are within a circle of radius 5 centered at the origin. If so, a retinal cell is copied from the library to the */retina/recplane/rec[]* array and given *x-y* coordinates that are scaled versions of *i* and *j* using the *position* command. The *position* command, as its name suggests, sets the position coordinates of an element to equal its last three arguments (which are *x-value*, *y-value*, *z-value*, respectively). You may wish to check that the above function does in fact arrange the cells in a circular array with a diameter of 200 microns centered on the origin.

This function also illustrates another feature of GENESIS programming: if there is no built-in command to perform a particular task, you can usually write a script function to do it. If there is a built-in command, however, it is nearly always more efficient (i.e., faster) than an equivalent script function.

Now we return to the actual simulation. The statement

```
showfield /retina/recplane/rec[0] -all
```

reveals (among other things)

```
xyz      = ( -2.000000e-04 , -2.000000e-04 , 0.000000e+00 )
```

This means that the *x*, *y*, *z* positions of the cell are $(-200, -200, 0)$ (in microns). Likewise,

```

rec[1] --> xyz = ( -1.600000e-04 , -2.000000e-04 , 0.000000e+00 )
rec[5] --> xyz = ( 0.000000e+00 , -2.000000e-04 , 0.000000e+00 )
rec[9] --> xyz = ( 1.600000e-04 , -2.000000e-04 , 0.000000e+00 )
rec[54] --> xyz = ( -4.000000e-05 , 0.000000e+00 , 0.000000e+00 )
rec[55] --> xyz = ( 0.000000e+00 , 0.000000e+00 , 0.000000e+00 )

```

This shows that all the retinal cells have the same z coordinate (0) but are positioned differently in x - y space.

In a similar manner, the script *V1.g* makes 25 copies of the V1 cell in */V1/horiz* with the statement:

```

createmap /library/soma /V1/horiz \
{V1_NX} {V1_NY} \
-delta {V1_SEPX} {V1_SEPY} \
-origin {-V1_NX * V1_SEPX / 2} {-V1_NY * V1_SEPY / 2}

```

Executing the command “`le /V1/horiz`” gives “`soma[0-24]/`”; again, the final “`/`” indicates that the array of cells have subelements connected to them. This array is 5 by 5 with spacings of 80×10^{-6} m (*V1_SEPX* and *V1_SEPY*) in the x and y directions. These represent the V1 cells selective to horizontally oriented stimuli. A similar statement is used to create the plane of vertically selective cells (called */V1/vert*). All of the cells in both planes have z coordinates of 0.0. In this case the spatial coordinates of the horizontal and vertical cells overlap, but this will cause no problems since the two groups of cells can be referred to and manipulated separately.

At this point we have three arrays of cells, representing the source “retinal” cells, and the vertically and horizontally selective destination “V1” cells. The next step is to set up connections between these cells to form a network.

18.7 Making Synaptic Connections

There are two different ways of specifying the nature of connections between cells in GENESIS. We can individually specify each connection and its associated parameters, or we can use special GENESIS commands that define and parameterize groups of connections between groups of elements. The *Orient.tut* simulation employs the second approach, using the commands *planarconnect*, *planardelay*, and *planarweight* (in the script file *ret_V1.g*). First, though, we will remind you how we manually set up connections in Chapter 15. This approach might be feasible in modeling a system with precise connections between identified cells, such as some invertebrate systems and the central pattern generator circuits treated in Chapter 8. For larger networks with less precise connectivities, the commands operating on groups of elements and connections are more appropriate.

A synaptic connection in GENESIS is made between a spike generating object (such as **randomspike** or **spikegen**) and a synaptic channel object (such as **synchan**). Axons as such are not modeled explicitly (although there is an obsolete **axon** object that is used in some older simulations). In general, the somatic compartment of a cell will be linked to a **spikegen** object, which will detect when a spike has occurred and pass that information along to the synaptic channel object. The V1 cells in our case are set up in this way, although in this simple model the V1 cells are not connected to anything (but see the exercises at the end of the chapter). Alternatively, a **randomspike** object can be used to provide randomly occurring spikes at a given frequency to the synaptic channel object. The retinal cells in our model are modeled as **randomspike** objects, and we impose a firing pattern on the retina by manipulating the firing rates of the retinal cell directly (discussed in Sec. 18.8).

18.7.1 Specifying Individual Synaptic Connections

The connection between the spike generating object and the synaptic object is established by adding a message between the two objects. For instance, to connect the retinal cell `/retina/recplane/rec[0]` with an excitatory synapse on the cell `/V1/horiz/soma[0]`, we could use the following command.

```
addmsg /retina/recplane/rec[0]/input /V1/horiz/soma[0]/exc_syn SPIKE
```

Here, the presynaptic element is a **randomspike** object and the postsynaptic element is a **synchan** object, or synaptically activated channel, discussed in Chapter 15. Since the synaptic connection presumably includes a time delay between the time the spike occurred and the time its influence is felt on the synaptic channel, due to both the axonal and synaptic transmission delays, we have to specify that delay explicitly (the default is a delay of 0, which is not very realistic). Also, the effects of different synapses on their postsynaptic targets will differ in magnitude. This is modeled by a weight field on the synapse. Assuming that the above SPIKE message created synapse number 0 on the **synchan** object, the weight and delay may be set as follows.

```
setfield /V1/horiz/soma[0]/exc_syn synapse[0].weight 2.0 \
synapse[0].delay 1e-4
```

for a weight of 2.0 and a delay of 0.1 msec (10^{-4} sec). It should be emphasized that synaptic “weight” is a dimensionless field: a weight of 1.0 means that a single spike will cause a conductance change with a maximum height of *gmax*, the “maximal” conductance of the synapse. However, a weight of 2.0 will cause a conductance change that is twice as large. Thus, the *gmax* field in the **synchan** object refers to the maximal conductance of the channel when the synaptic weight is 1.0.

To make things even easier on us, we could alternatively write a script-level command to set up synapses as follows.

```

function makesynapse(pre,post,weight,delay)
    str pre, post
    float weight, delay
    int syn_num
    addmsg {pre} {post} SPIKE
    syn_num = {getfield {post} nsynapses} - 1
    setfield {post} synapse[{syn_num}].weight {weight} \
                synapse[{syn_num}].delay {delay}
end

```

When we add a SPIKE message, the last synapse corresponds to the message just added. We have to set *nsynapses* to (*nsynapses* – 1) since synapses are numbered starting with 0. Once this function is defined, setting up a synapse is as easy as typing:

```
makesynapse /retina/recplane/rec[0]/input /V1/horiz/soma[0]/exc_syn \
2.0 1e-4
```

We've just shown you how to set up synapses individually in GENESIS. However, in any reasonably sized network simulation, it would be extremely tedious to set up all the connections in this way. We could use a *for* loop along with the *makesynapse* function defined above, but there is an easier way. GENESIS includes several commands that can be used to set up large numbers of synapses all at once, thus simplifying the process of setting up network-level simulations. This is the approach taken in the *Orient_tut* simulation, and is described next.

18.7.2 Commands Involving Groups of Synapses

Connecting Groups of Synapses

The connections between the retina plane and the two *V1* planes are made in *ret_V1.g* with statements like:

```

planarconnect /retina/recplane/rec[]/input \
               /V1/horiz/soma[]/exc_syn \
               -relative \
               -sourcemask box -1 -1 1 1 \
               -destmask box {-V1_SEPX * 2.4} {-V1_SEPY * 0.6} \
                           { V1_SEPX * 2.4} { V1_SEPY * 0.6}

```

The purpose of this rather complex command is to connect a region of identical elements to another region of identical elements (the elements of the second region are not necessarily the same kind of elements as those of the first region). The *planar* in *planarconnect* refers to the fact that the source elements are viewed as lying in a two-dimensional plane. Since

GENESIS objects can have three-dimensional locations, this means that only the *x* and *y* dimensions are used in this command. The command is intended to be used for elements located in a two-dimensional sheet with a constant *z* value, which is true for objects created with the *createmap* command described above. The full usage for this command is:

```
planarconnect source_elements destination_elements \
  [-relative] \                                // relative or absolute connection_mode
  -sourcemask {box,ellipse} \                  // elliptical or rectangular region
  x1 y1 x2 y2 \                            // range of source cells
  [-sourcehole {box,ellipse} \                // range of source cells not to connect
  x1 y1 x2 y2] \
  -destmask {box,ellipse} \                  // range of destination cells
  x1 y1 x2 y2 \                            // range of dest cells not to connect
  [-desthole {box,ellipse} \                // range of dest cells not to connect
  x1 y1 x2 y2] \
  [-probability p]                         // probability of making connection
```

Incidentally, notice that in GENESIS commands that span several lines and that are continued using backslashes (\) you can include comments after the backslashes. The empty brackets (*rec[]* and *soma[]*) indicate that all of the **spikegen** elements in the retinal cells (i.e., */retina/recplane/rec[0-99]/input*) will be connected to the **synchan** elements in */V1/horiz/soma[0-24]/exc_syn*. “-relative” means that the (*x*, *y*) coordinates of the destination elements will be measured relative to those of the source elements. The default is to use the absolute coordinates of the destination elements. The **-sourcemask** option specifies the range of source elements to connect, as either a rectangle (box) in the *x*-*y* coordinate space of the elements or an ellipse. For a rectangular (box) region, the coordinates *x1* and *y1* refer to the minimum *x* and *y* values of the rectangular region, and the coordinates *x2* and *y2* refer to the maximum *x* and *y* values of the rectangular region. For an elliptical region, *x1* and *y1* are the coordinates of the center of the ellipse whereas *x2* and *y2* are the lengths of the principal axes in the *x* and *y* directions, respectively. The curly braces mean you have to choose one or the other of {*box*, *ellipse*}.

In this case, the coordinates -1 -1 1 1 for the **sourcemask** span a region far larger than the total extent of the source region (2 meters square), so that all source elements will be connected. You can specify multiple source regions by specifying several lines of the form **-sourcemask {box,ellipse} x1 y1 x2 y2**. The **-sourcehole** option indicates a range of elements not to connect; this is useful when you want to connect all elements in a rectangular region except for some inside the region (see below for examples). Again, you can specify multiple “holes” if you want. The **-destmask** and **-desthole** options similarly specify the coordinates of the destination elements. Finally, the **-probability** option specifies the probability of connections, which is 1.0 by default (all elements in the source region(s) specified are connected with all elements in the destination region(s) specified).

The way this works in practice is as follows. GENESIS looks at the list of source elements and rejects those not in the source region. For each source element within the source region, it scans the list of destination elements and picks out those whose position is in the destination region (measured either in absolute coordinates or relative to the specific source element, if the `-relative` option has been selected). Then it makes a synaptic connection between the source and destination elements. If the `-probability` option has been selected the connection will be made with the given probability, so not all possible connections will be made.

Confused? Here are some examples. First, say we wanted the source region to consist of all the `rec` cells except for a rectangular region of 20 microns square in the middle, with the destination cells the same as above. Then you would use the command:

```
planarconnect /retina/recplane/rec[]/input \
    /V1/horiz/soma[]/exc_syn \
    -relative \
    -sourcemask box \
    -1 -1 1 1 \
    -sourcehole box \
    -20e-6 -20e-6 20e-6 20e-6 \
    -destmask box \
    {-V1_SEPX * 2.4} {-V1_SEPY * 0.6} \
    { V1_SEPX * 2.4} { V1_SEPY * 0.6} // range of dest region
```

Alternatively, say we wanted to have two destination regions for the connections between the receptors and the vertically selective V1 cells, one of which includes all the cells whose *x* coordinates are between 10 and 20 μm less than the receptor cells' *x* coordinates and one of which includes all the cells whose *x* coordinates are between 10 and 20 μm more than the receptor cells' *x* coordinates. Also suppose we wanted to exclude a circular part of the source region centered at the origin and 20 μm in diameter, but otherwise include all the source cells. Then we could write this:

```
planarconnect /retina/recplane/rec[]/input \
    /V1/vert/soma[]/exc_syn \
    -relative \
    -sourcemask box \
    -1 -1 1 1 \
    -sourcehole ellipse \
    0 0 20e-6 20e-6 \
    -destmask box \
    -20e-6 -1 -10e-6 1 \
    -destmask box \
    10e-6 -1 20e-6 1
```

// range of source region -- all cells
 // circular region centered at origin
 // range of first dest region
 // range of second dest region

Of course, these examples are just to illustrate the options available; we don't claim that these connection patterns are ideal for generating good orientation selectivity.

There is also a three-dimensional analog to *planarconnect*, which we'll mention here for completeness even though it isn't used in *Orient_tut*. It is *volumeconnect*, with the following usage.

```
volumeconnect source_elements destination_elements \
  [-relative]
  -sourcemark {box,ellipsoid} x1 y1 z1 x2 y2 z2
  [-sourcehole {box,ellipsoid} x1 y1 z1 x2 y2 z2]
  -destmask   {box,ellipsoid} x1 y1 z1 x2 y2 z2
  [-desthole   {box,ellipsoid} x1 y1 z1 x2 y2 z2]
  [-probability p]
```

The syntax is exactly the same as *planarconnect*, except that x_1 , y_1 , and z_1 refer to the minimum x , y , and z coordinates while x_2 , y_2 , z_2 refer to the maximum coordinates for a box region; for an ellipsoid region x_1 , y_1 , and z_1 are the coordinates of the center of the region while x_2 , y_2 , z_2 are the lengths of the principal axes in the x , y and z regions, respectively.

NOTE: if, through an error in syntax, you mistakenly specify source or destination elements that don't exist, no error message will be given. Therefore, it is a good idea to check to see if the connections exist. One way to find out what synaptic connections exist is to use the following command:

```
showmsg /retina/recplane/rec[54]/input
```

This refers to the **randomspike** element called *input*, which is part of receptor cell 54. The output gives the messages sent from this element:

```
MSG 0 to '/V1/horiz/soma[10]/exc_syn' type [-1] 'SPIKE'
MSG 1 to '/V1/horiz/soma[11]/exc_syn' type [-1] 'SPIKE'
MSG 2 to '/V1/horiz/soma[12]/exc_syn' type [-1] 'SPIKE'
(etc.)
```

This shows that the receptor is connected to the excitatory synapses of the somata of several cells. The elements receiving the messages are **synchan** objects. If you have a lot of messages being passed from the source element aside from the SPIKE message, you can type

```
showmsg /retina/recplane/rec[54]/input | grep SPIKE
```

which will only display the SPIKE messages. The *showmsg* command doesn't tell you what the weight or delay is for each connection; that information is stored in the **synchan** objects (on the postsynaptic side). Likewise, you can check that

```
showmsg /retina/recplane/rec[55]/input | grep SPIKE
```

shows targets in the V1 horiz layer of somas 11–19, not including 15.

On the postsynaptic side, we can use the *showmsg* command to display the connections (SPIKE messages) coming into a synapse (**synchan** object). For example, if we type

```
showmsg /V1/horiz/soma[24]/exc_syn | grep SPIKE
```

we get

```
MSG 1 from '/retina/recplane/rec[74]/input' type [-1] 'SPIKE'  
MSG 2 from '/retina/recplane/rec[75]/input' type [-1] 'SPIKE'  
MSG 3 from '/retina/recplane/rec[76]/input' type [-1] 'SPIKE'  
MSG 4 from '/retina/recplane/rec[77]/input' type [-1] 'SPIKE'  
(etc.)
```

See Sec. 18.7.3 for a more comprehensive way of obtaining information about synaptic connections.

Setting the Delay Fields of Groups of Synapses

The transmission delays are set with the *planardelay* function. In this simulation, the command

```
planardelay /retina/recplane/rec[]/input -radial {CABLE_VEL}
```

uses the *x* and *y* coordinates to calculate the radial distance from each of the source elements (*rec[0]/input* through *rec[99]/input*) to each target for the synaptic connections. This distance is divided by the scale factor, *CABLE_VEL*, in order to assign a value for the delay field of the axon connection. *CABLE_VEL* stands for the velocity of axonal propagation, in *m/sec*. Dividing the distance between two objects by the propagation velocity gives the time delay for a spike occurring at one cell to reach the postsynaptic cell, assuming that the axons are oriented radially. Note that the distance between the two planes does not enter into this calculation. Also note that source elements for this command must be derived from objects that can send SPIKE messages, which usually means **randomspike** objects or **spikegen** objects.

The full syntax for the *planardelay* function is:

```
planardelay sourcepath  
[-fixed delay]  
[-radial conduction_velocity]  
[-add]  
[-uniform scale]
```

```
[-gaussian stdev maxdev]
[-exponential mid max]
[-absoluterandom]
```

There are several options for the *planardelay* function. The first two options (*-fixed* and *-radial*) are mutually exclusive. *-fixed* means that the delays from the source are all nominally equal to *delay*. *-radial* means that the delays from the source are scaled according to the radial distance between the source and the targets. *conduction_velocity* represents the conduction velocity of the spike along the (hypothetical) axon. As mentioned above, the computed delay between two elements equals the radial distance between the elements (computed by the function) divided by the conduction velocity. The *-add* option causes the delays computed using either the *-fixed* or *-radial* commands to be added to the preexisting delay (the default is to simply replace the existing delay with the new value). This can be useful if you are modeling cells connected by fiber tracts that have sections with different conduction velocities, for example, when fast-conducting axons give rise to slower-conducting axon collaterals. In that case you can call *planardelay* once to set up the delays from the axon and call it again using the *-add* option to add the delays from the axon collaterals using a different conduction velocity.

The other options represent ways of adding random components to the delays. Since these same options are used for several commands, they are discussed in further detail later in this section under “**Adding Randomness to Weights and Delays**.”

There is also a three-dimensional analog of this command, called *volumedelay*, with the same syntax,

```
volumedelay path
  [-fixed delay]
  [-radial conduction_velocity]
  [-add]
  [-uniform scale]
  [-gaussian stdev maxdev]
  [-exponential mid max]
  [-absoluterandom]
```

The only difference between *planardelay* and *volumedelay* is that *volumedelay* calculates the radial distance using all three dimensions instead of just the *x* and *y* dimensions.

There is also a separate command called *syndelay*, for adding a small synaptic component to the delays. This is useful when cells are very close together and the delay calculated using the *-radial* option of *planardelay* or *volumedelay* is unrealistically small. The usage of this command is:

```
syndelay path delay
  [-add]
```

```

[-uniform scale]
[-gaussian stdev maxdev]
[-exponential mid max]
[-absoluterandom]

```

In this case the path specification is to a group of postsynaptic objects (i.e., **synchan** elements). The delay is equal to the “delay” argument of the command, with the appropriate random component added. If you want to add this delay to a delay previously determined using *planardelay*, say, use the *-add* option as with *planardelay*. If not, the computed delays become the delays of the synapses and if you want to add axonal delays you will have to use *planardelay* or *volumedelay* with the *-add* option. In general, one usually sets the axonal delay first and then adds on the synaptic delay if desired. It is important to note that the synaptic delay is NOT a separate field in the synapse; the axonal and synaptic delays are lumped together in a single “delay” field.

Setting the Weight Fields of Groups of Synapses

Finally, we have to assign weights to the synapses, since the *planarconnect* function initializes all weights to zero. In the *Orient_tut* simulation, this is done with the command

```
planarweight /retina/recplane/rec[]/input -fixed 0.22
```

This command is of the form

```

planarweight sourcepath
  [-fixed weight]
  [-decay decay_rate max_weight min_weight]
  [-uniform scale]
  [-gaussian stdev maxdev]
  [-exponential mid max]
  [-absoluterandom]

```

where *sourcepath* normally refers to the **spikegen** or **randomspike** elements of the source cells. In this command, the first options (*-fixed* and *-decay*) are mutually exclusive and determine whether the weights fall off with distance from the source. The *-fixed* option makes all weights from that sourcepath nominally equal to *weight*. The *-decay* option works like *planardelay* calculating a radial distance between the source elements and each target. The parameter *decay_rate* gives the rate for an exponential decay of the weights with radial distance. Note that *decay_rate* has the units of *meters⁻¹*. The function describing the weight as a function of *max_weight* and *min_weight* for this option is

$$\begin{aligned} \text{weight} = & (\text{max_weight} - \text{min_weight}) \times \exp(-\text{decay_rate} \times \text{radial_distance}) \\ & + \text{min_weight}. \end{aligned}$$

The reason one might want weights that decay exponentially with distance depends on the conceptual framework of the simulation. If each synapse in your network simulation is intended to represent one synapse in the real system, then the weights could have any (physiologically reasonable) value, and it might be best to set them to a random value within reasonable limits if you had no *a priori* reason to set them to particular values for particular cells. On the other hand, since a simulator typically models a large network of neurons with a much smaller number of simulated cells, each cell can be thought of as representative of a group of cells in a particular region. In this case, it makes sense that, on average, simulated cells close together will have stronger connections between them (i.e., synapses with larger weights) than simulated cells located farther apart from each other, for the simple reason that the real cells of which the simulated cells are representative will in general form more connections between them if they are close together than if they are far apart. It has to be kept in mind that unless you intend to model every cell in the network, each cell is really an abstraction of a class of cells, and the strength of the synapses has to reflect the nature of this abstraction. If you don't want exponential decay of weights, you should use the `-fixed` option. In the *Orient_tut* simulation we use the `-fixed` option since we aren't modeling connections between V1 cells (but see the exercises at the end of the chapter). Remember that if you don't use *planarweight* (or *volumeweight*, described next), you have to set the weights explicitly, since they are equal to zero by default.

As usual, there is also a three-dimensional analog of this command, called *volumeweight* with the same syntax and function:

```
volumeweight sourcepath
  [-fixed weight]
  [-decay decay_rate max_weight min_weight]
  [-uniform scale]
  [-gaussian stdev maxdev]
  [-exponential mid max]
  [-absoluterandom]
```

The only difference between *volumeweight* and *planarweight* is that, for the `-decay` option, the distances are calculated using all three dimensions instead of just the *x* and *y* dimensions.

Adding Randomness to Weights and Delays

The above commands for setting weights and delays have a set of options for adding a random component to the weights and delays set. These options are the same for all these commands, and have the form:

```
<command>
<command-specific options>
```

```
[-uniform scale]
[-gaussian stdev maxdev]
[-exponential mid max]
[-absoluterandom]
```

Each of the first three options selects a random number out of a particular probability distribution. The *scale* option of `-uniform` gives a random number uniformly distributed in the range $\{-\text{scale}, \text{scale}\}$. The `-gaussian` option gives a Gaussian-distributed random value with a mean of zero, a standard deviation of *stdev*, and a maximum deviation of *maxdev*. The `-exponential` option gives an exponentially distributed value with a minimum value of zero, $1/e$ point (i.e., the point at which the probability density function has decayed to $1/e$ of its maximum value) at *mid* and maximum value of *max*. The *max* or *maxdev* arguments are useful in cases where you want to truncate the ends of a distribution to prevent weights and delays from being larger or smaller than some limit. For instance, a suitable choice of *max* or *maxdev* will prevent the possibility of setting weights or delays to a negative value, which is biologically meaningless. However, as an added precaution, the commands for setting groups of weights and delays will set negative values that may arise from using the random options to zero.

The way these options are used is as follows. First the weight or delay is calculated according to the command-specific options of the command. Let's say that *val* is the value of a weight or delay before any randomness is added. After the random number is included, we have

$$\textit{val} = \textit{val} + (\textit{val} \times \textit{random_number})$$

unless the `-absoluterandom` option is used, in which case we have

$$\textit{val} = \textit{val} + \textit{random_number}$$

In other words, normally the value of the random number is scaled to the existing weight or delay before adding it to the weight or delay. This is reasonable in most cases, since you usually want to add a certain proportion of variability to all weights or delays. Thus it's easy to add, say, up to 10% randomness to your weights — just use `-uniform 0 .1`. If you want to add random numbers from the same distribution to all weights or delays regardless of their original size, use the `-absoluterandom` option (which you can abbreviate as `-abs`).

In all these cases, the random number is chosen separately for each synaptic connection. The command “`randseed <number>`” will initialize the random number generator with *number*. If `randseed` is called with no arguments it will initialize the random number generator with the current time, giving random numbers that will be different each time you run the simulation.

Here are a couple of examples. In the above case, if you wanted to have delays corresponding to conduction velocities uniformly distributed between 1 and 2 *m/sec* (i.e.,

$1.5 \pm 0.5 \text{ m/sec}$, or $1.5 \pm 33\%$) you could type

```
planardelay /retina/recplane/rec[]/input -radial 1.5 -uniform 0.33
```

(We're assuming you're using SI units here.) Note that the scaling is important here, since the delays are calculated by dividing the distance by the conduction velocities. Thus, delays corresponding to nearby elements will be shorter than those corresponding to elements separated by a greater distance. Therefore, it is important that the random component of the delay be scaled to a value that is a constant proportion of the total delay.

If you wanted weights normally distributed with a mean of 2.0, a standard deviation of 10% of the mean (i.e., 0.2), and a maximum deviation of 40% of the mean (0.8), giving the weights the range of {1.2, 2.8}, you could type

```
planarweight /retina/recplane/rec[]/input -fixed 2.0 -gaussian 0.1 0.4
```

In this case, we could have used the `-absoluterandom` option with the arguments `-gaussian 0.2 0.8` to get the same effect, since here the weights are all the same at the beginning. It is very important to bear in mind that the arguments to the options involving randomness are relative to the actual weight or delay value unless the `-absoluterandom` option is used.

After setting up weights and delays with the above commands, it would be a good idea to check that they are in the desired ranges. We could use `showfield` for this, but it would be a tedious procedure to track down all the connections and to inspect the various `synchan` fields. Fortunately, GENESIS has some commands that make this easier.

18.7.3 Utility Functions for Synapses

In general, we do not want to be concerned with the synapse number (hereafter called the synapse *index*, since it represents the index of an array of synapses) when setting up weights and delays. In addition, sometimes we may need to access other information about synapses, such as the source element of a given synapse or the total number of synapses. GENESIS provides several utility functions for this purpose, which we describe here. These functions will be particularly useful for debugging a simulation. Although the commands described above are much more efficient than the use of *for* loops for the establishment of network connections, it is easy to make a mistake in syntax and not get the connections that were intended.

The function `getsyncount` has the usage

```
getsyncount [presynaptic-element] [postsynaptic-element]
```

This function is used to count synapse numbers. Either one or both options must be specified. The most common usage is to specify only the presynaptic element. In this case, it returns the number of SPIKE messages that are sent by that element. If only the postsynaptic element (e.g., a **synchan**) is present, it returns the number of synapses in that element. As we have seen in Chapter 15 and in Sec. 18.7.1, we could also obtain this result by using *getfield* to retrieve the *nsynapses* field of the postsynaptic element. If both arguments are present, it returns a count of the number of synapses in the postsynaptic element that receive SPIKE messages from the presynaptic element. (This will almost always return 0 or 1, as redundant connections between the same source and destination are more efficiently handled by increasing the synaptic weight of the connection.)

The function *getsynindex* has the usage

```
getsynindex <presynaptic-element> <postsynaptic-element> [-number n]
```

It is used to find the index of synapses between the given presynaptic and postsynaptic elements. The *-number* option will give the index of the *n*th synapse between the presynaptic and postsynaptic target. This option should rarely be necessary, since usually there is at most one synapse between a given presynaptic and postsynaptic element. If no matching synapse is found, a warning message is printed and the function returns *-1*.

We can use the *getsynindex* function in GENESIS to help us set the weights and delays of a synapse whose presynaptic element is known but whose index is not. For example we could type

```
int syn_num = {getsynindex /retina/recplane/rec[0]/input \
/V1/horiz/soma[0]/exc_syn}
setfield /V1/horiz/soma[0]/exc_syn synapse[{syn_num}].weight 2.0 \
synapse[{src}].delay 1e-4
```

The function *getsynsrc* has the usage

```
getsynsrc <postsynaptic-element> <index>
```

This function returns a string that is the path of the presynaptic element sending the SPIKE message to the synapse of the postsynaptic element with the given index.

The function *getsyndest* has the usage

```
getsyndest <presynaptic-element> <n> [-index]
```

This function returns a string that is the path of the postsynaptic element which receives the *n*th SPIKE message sent by the presynaptic element. The *-index* option returns the index of the synapse corresponding to this message. Alternatively, after having found the destination synapse, you may find its index by using *getsynindex*.

As an example of the use of the above functions, we can write a script function to give information about all the synapses in a particular **synchan**:

```
function synapse_info(path)
    str path, src
    int i
    float weight, delay
    floatformat %.3g
    for(i = 0; i < {getsyncount {path}}; i = i + 1)
        src  = {getsynsrc {path} {i}}
        weight = {getfield {path} synapse[{i}].weight}
        delay = {getfield {path} synapse[{i}].delay}
        echo synapse[{i}]: \
            src = {src} weight = {weight} delay = {delay}
    end
end
```

This function also uses the *floatformat* command to set the format of the output to *%.3g*, which displays at most three significant figures and rounds the output to reasonable values. The GENESIS default is to print out 10 significant digits, which is often unnecessary.

We can use this function to check the ranges of the weights and delays of the synapses in the *Orient_tut* simulation. For example,

```
synapse_info /V1/horiz/soma[12]/exc_syn
```

gives the output

```
synapse[0]: src = /retina/recplane/rec[30]/input weight = 0.22 delay = 0.000165
synapse[1]: src = /retina/recplane/rec[31]/input weight = 0.22 delay = 0.000126
synapse[2]: src = /retina/recplane/rec[32]/input weight = 0.22 delay = 8.94e-05
synapse[3]: src = /retina/recplane/rec[33]/input weight = 0.22 delay = 5.66e-05
synapse[4]: src = /retina/recplane/rec[34]/input weight = 0.22 delay = 4e-05
(etc.)
```

18.8 Setting Up the Inputs

After the network is constructed, we will usually want a way to provide some input to the network. The details of this will generally be specific to your simulation. The inputs to the *Orient_tut* network are specified in the file *ret_input.g*. This file defines several functions whose purpose is to sweep a vertical or horizontal bar across the retinal cells. This is done by imposing an average firing rate on these cells by setting the *rate* field of the **randomspike** elements in */retina/recplane/rec[0-99]/input*. There are a number of ways to accomplish this in GENESIS. One possibility would be to set up a *for* loop that runs through all the elements and sets the rate to a high or low value depending on the position of the cell in space. Another approach, which is used in the file, is to define a function called *do_autosweep* which is invoked on each step of the simulation. Yet another approach would be to define a GENESIS extended object that performs the same function. More information is given in the *README* file in the *Orient_tut* directory and in the comments in the *ret_input.g* file.

18.9 Summary

In this chapter we have shown you the GENESIS commands for creating groups of cells and connecting them in networks. We have discussed how to set up synapses individually and also have described various commands that allow you to set up groups of synapses simultaneously. These commands enable you to configure the connectivity patterns, synaptic weights, and synaptic delays of a network in very flexible ways. We have used the *Orient_tut* simulation as an example to show how these commands are used in a real simulation. The *Orient_tut* simulation also implements a graphical user interface that allows the user to look at various aspects of the network as it is being simulated, including the firing patterns of the input (retinal) cells, the membrane potentials of the output (V1) cells, and the connection patterns in the network. The objects and commands used to set up the interface are described in Chapter 22.

18.10 Exercises

1. Verify that with `CABLE_VEL = 1`, the delays for the connections from `rec[54]/axon` to the targets in the V1 horiz layer of somas 11, 12 and 13 are correct (i.e., are equal to the radial distances).
2. Modify the “`synapse_info`” function, using the synaptic utility functions described previously, to generate the pathname, synapse index, weight, and delay values of all synapses projecting from a given **randomspike** element, i.e., all synapses receiving SPIKE messages from that element. You might want to save this function for later use.
3. Look at the file `ret_V1.g`. How is the orientation-selectivity conferred on the network? Can you improve the selectivity just by changing some parameters of the commands?

The cells in the two V1 planes contain some elements that are not used in the simulation. The spike generators of the cells connect to nothing and there is no input to the inhibitory channels, `inh_syn`. As an exercise in using the commands discussed above,

4. Modify the `ret_V1.g` script by adding synapses between the retinal cells and the inhibitory synapses of the V1 cells in order to improve the quality of the orientation selectivity.
5. Modify the `ret_V1.g` script to generate connections between V1 cells. Can you use the feedback connections to further improve the quality of the orientation selectivity?

Chapter 19

Implementing Other Types of Channels

DAVID BEEMAN

19.1 Introduction

So far, we have been using “squid-like” Hodgkin–Huxley channels for our voltage-dependent channels. In Chapter 7, you were introduced to a much wider variety of ionic conductances. There are scripts that contain functions for creating these and many other channel models in the *Scripts/neurokit/prototypes* directory. If you are constructing a realistic cell model and are lucky, you will find a function to create the channel you need in one of these scripts. Sometimes you may need to write your own function or, at least, modify a function that is similar to the one you need.

In Chapter 14, we learned how to implement voltage-dependent channels with the **hh_channel** object. This object is fairly easy to use, and has a straightforward correspondence between its internal fields and the parameters used in the Hodgkin–Huxley model. However, there are several good reasons for preferring another GENESIS channel object, the **tabchannel**. Although this object solves differential equations of the Hodgkin–Huxley form, given in Eq. 14.3, the rate parameters are provided by a table lookup, rather than from a fit to one of the three functional forms (Eqs. 14.4 – 14.6) used by the **hh_channel**. For further flexibility, GENESIS offers a related object, the **tab2Dchannel**, which uses two-dimensional tables.

Some channel models use expressions for the rate constants α and β that are not in one

of these forms. Even if the rate constants can be expressed in this manner, it is much faster to use a table lookup than to evaluate the functional form of the rate variable. Thus, a model that contains a great many channels will run much faster if it uses **tabchannels** or **tab2Dchannels**, rather than **hh_channels**. In addition, the **tabchannel** and the **tab2Dchannel** may be used with the fast implicit numerical integration methods discussed in Chapter 20. Not only will these methods further increase the speed of your simulation, but they are required in order to obtain numerically stable and accurate solutions for large cell models that contain many compartments. For these reasons, the **tabchannel** and the **tab2Dchannel** are the preferred objects to use for implementing voltage-dependent channels in large models.

In the following sections, we use the **tabchannel** to model channels from experimental data that have not been fitted to equations. We use a similar procedure to implement models for rate parameters that are not in one of the three standard forms (Eqs. 14.4–14.6). With the **tabchannel**, **tab2Dchannel** and some other GENESIS objects, we model channels that have a conductance that depends on the intracellular concentration of calcium ions. The **synchan** object, introduced in Chapter 15, provides a fairly general way to implement most synaptically activated channels. Towards the end of this chapter we discuss other approaches that may be used to implement NMDA channels, gap junctions, and dendrodendritic synapses.

In this chapter, we give examples of script language functions that may be used to create various types of channels. These are in a form suitable for use as prototypes with current versions of *readcell* (Chapter 16) and *Neurokit* (Chapter 17).

19.2 Using Experimental Data to Make a **tabchannel**

Smith and Thompson (1987) used voltage clamp experiments similar to those described in Chapter 4 to measure the characteristics of the slow inward tail current (I_B) that is found in bursting pacemaker cells of *Tritonia diomedea*. This current is often called the “B-current” because it is believed to be responsible for maintaining the prolonged depolarization that allows bursts of action potentials to occur. We will use results taken from this paper to construct a channel model, using the **tabchannel** object. This model was used in the simulation of the bursting molluscan neuron in Chapter 7. The GENESIS functions that implement this channel model and the others that are used in the simulation can be found in the script *ASTchan.g* in the *neurokit/prototypes* directory.

Figure 19.1A replots the Smith and Thompson data for the experimentally measured time constant (squares). As is typical with these sorts of measurements, there is a large amount of experimental uncertainty in these values, as well as variation from sample to sample. Therefore, we will want to draw a smooth curve to fit the data points, as shown in the dotted line, and take our values from this curve. Although a curve-fitting program could be used here, an “eyeball” fit by hand is adequate, considering the amount of noise in the data. Ideally, we would like to have similar data for the steady-state value of the activation

state variable, m_∞ . As is often the case, we are only given the steady-state current

$$I_B = \bar{g}_B m^p (E_{rev} - V). \quad (19.1)$$

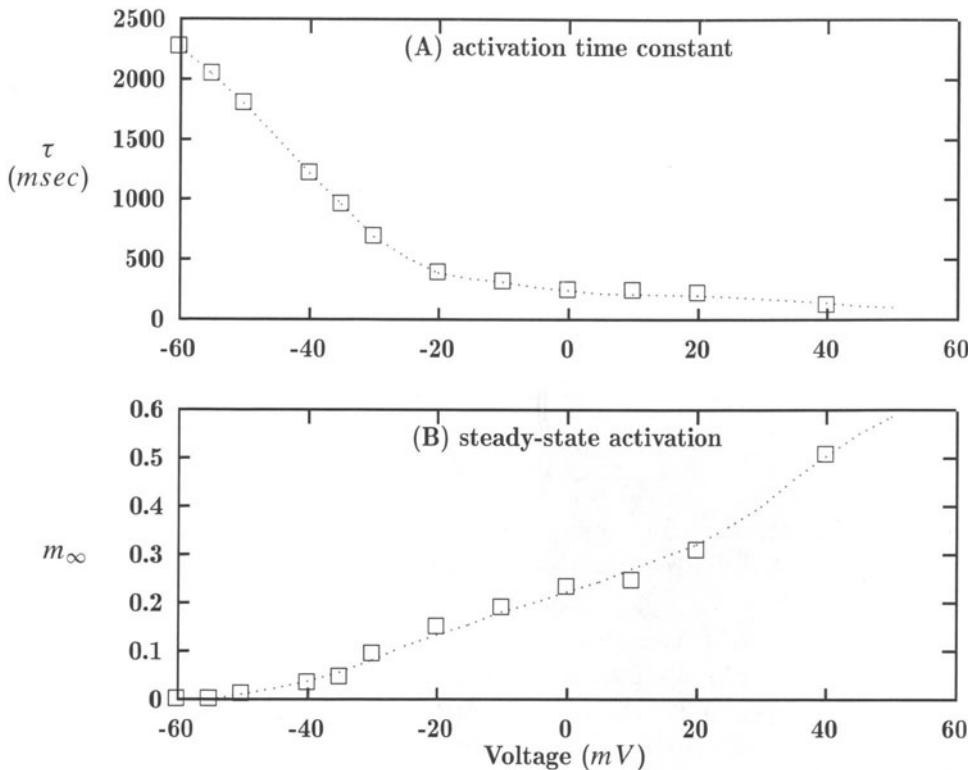


Figure 19.1 Data taken from Smith and Thompson (1987) for the B-current time constant (A) and steady-state activation (B). The squares represent experimental data and the dotted lines represent a fit to the data.

They estimated E_{rev} to be approximately 68 mV, but did not perform any fitting to determine the exponential p , or the maximum conductance \bar{g}_B . When fitting m_∞ to a sigmoid or other analytical function, it is customary to choose a value of the power p that gives the best fit, as discussed in Sec. 4.4.1 and illustrated in Fig. 4.5. In this case, we have used $p = 1$. The maximum conductance \bar{g}_B is also unknown. Often, we can estimate it by using Eq. 19.1 to calculate and plot $\bar{g}_B m^p$ and making use of our expectation that m_∞ will asymptotically approach 1.0 at large values of V . This plot, shown in Fig. 19.1B, shows no sign of having reached a maximum, although there is some hint of sigmoidal behavior.

The best we can do is to guess that m has reached about half of its maximum value at 40 mV , and take \bar{g}_B to be about $0.1\text{ }\mu\text{S}$. This uncertainty will affect the scaling of the activation parameter, but will not affect any calculations using our channel model, as the channel current is the quantity of interest. By drawing a smooth curve through the data, as in Fig. 19.1B, we can estimate values of m_∞ to use in our channel model.

19.2.1 Setting the `tabchannel` Internal Fields

Like the `hh_channel`, the `tabchannel` has fields for E_k and G_{bar} , the activation state variables X and Y , and their associated exponents, X^{power} and Y^{power} . There is an additional variable Z and exponent Z^{power} that may be used to provide calcium concentration-dependent activation or inactivation. Thus, the channel conductance is calculated from an equation analogous to Eq. 14.1,

$$G_k = G_{bar} \cdot X^{X^{power}} Y^{Y^{power}} Z^{Z^{power}} \quad (19.2)$$

and the channel current is calculated from Eq. 14.2. Each of these three state variables obeys an equation of the form

$$\frac{dX}{dt} = A - BX. \quad (19.3)$$

In order to make the calculation more efficient, the notation is slightly different from that used in Eq. 14.3. A comparison of the two equations shows that $A = \alpha$ and $B = \alpha + \beta$. Each of the three gates (X , Y , and Z) can have associated tables to contain the voltage dependencies of the A and B variables.

We can illustrate the use of the `tabchannel` by constructing a script along the lines of `hhchan.g`, listed in Appendix B, which we used in Chapter 14. We will use the script to define a function to create a prototype channel that can be used with *Neurokit* and the cell reader, so the initial statements in the script will be fairly similar. Following the naming convention used with the other channel prototype scripts, we can call the channel `B_trit_st` and the function that creates it `make_B_trit_st`. As there is no inactivation, and we decided to let $p = 1$ in Eq. 19.1, the initial statements in our script should look something like

```
//genesis
float EB      = 0.068 // reversal potential in volts
float SOMA_A = 1e-9 // arbitrary value in sq m
function make_B_trit_st
    str chanpath = "B_trit_st"
    if (exists {chanpath})
        return
    end
    create tabchannel {chanpath}
```

```

setfield {chanpath}      \
Ek      {EB}           \
Gbar   {0.35*SOMA_A}  \
Xpower 1               \
Ypower 0               \
Zpower 0

```

At this point, the *hhchan.g* script contains statements for setting the parameters used to calculate α and β . In our script, we will create tables for *A* and *B* and fill them with data.

The **tabchannel** and other GENESIS objects that make use of tables keep the tabular data in a data structure called an *interpol_struct*. With some minor variations, the procedures for manipulating and accessing the contents of the *interpol_struct* are similar for all these objects. The first step is to allocate space for the tables in the *interpol_struct* and to set the values of fields that specify the number of table divisions (*xdivs*), the *x*-value corresponding to the first entry in the table (*xmin*), and that of the last (*xmax*). This is done by invoking the object's TABCREATE action with the *call* command. For a **tabchannel**, this is done with a command of the form

```
call <path> TABCREATE <gate> <xdivs> <xmin> <xmax>
```

where the “gate” is *X*, *Y*, or *Z*. Before writing the rest of your script, it would be a good idea to experiment with the **tabchannel** by entering some commands interactively to the GENESIS prompt. Try giving the commands

```

create tabchannel /foo
call /foo TABCREATE X 30 -0.100 0.050

```

This will create both an *A* table and a *B* table for the *X* gate. The two tables will be named *X_A* and *X_B* and will have identical values of *xdivs*, *xmin*, and *xmax*. The indices of the entries run from 0 to *xdivs*, so the *xdivs* field is literally the number of *intervals* in the table. The number of *entries* is *xdivs* + 1. The following commands, which you should try for yourself, illustrate the notation used for accessing these fields and table entries.

```

showfield /foo X_A->xdivs
showfield /foo X_A->xmin
showfield /foo X_B->xmax
setfield /foo X_A->table[0] 2.27
setfield /foo X_A->table[30] 0.104
showfield /foo X_A->table[0]
showfield /foo X_A->table[30]

```

Once the A and B tables have been set up and filled with the proper values, a VOLTAGE message can be sent from a compartment to the channel, giving the membrane potential to be used to calculate the channel conductance. If the voltage is -0.1 or less, the $table[0]$ values will be accessed. A voltage of 0.05 or greater will access the $table[30]$ values.

In principle, one would use tabulated values of m_∞ and τ in order to calculate the values of A and B that go into the tables. These would be given by

$$A = \alpha = m_\infty / \tau \quad (19.4)$$

$$B = \alpha + \beta = 1 / \tau. \quad (19.5)$$

However, GENESIS has a function, *tweaktau*, which allows us to fill the A table with τ and the B table with m_∞ . After the tables are filled, we can use this function to “tweak” the tables by performing the calculations given in Eqs. 19.4 and 19.5. There is an analogous function, *tweakalpha*, which would let us fill the tables with α and β and then invoke the function to refill them with the proper A and B values. We can make use of the interpolation capability of GENESIS tabular objects in order to minimize the number of data points that we need to enter into the tables. Data points taken at 0.005 V intervals will give us a fairly smooth curve. Given the uncertainty of the experimental data, no further precision is justified. The range of -0.1 to 0.05 volts used in the example above covers the membrane potentials we would expect to find. Setting *xdivs* to 30 gives us the desired voltage increment. The data in Fig. 19.1A begin at -60 mV, where the smoothed value of τ is 2.27 sec. Rather than trying to extrapolate to lower voltages, we will fill the first eight entries of the A table with this value. This can be done by setting each of these table entries individually, but there is an easier way.

The *neurokit/prototypes/defaults.g* script, which is included when *Neurokit* is run, defines a GENESIS script function, *settab2const*, to do this sort of thing. If you are not using *Neurokit*, you may include *defaults.g* in your simulation, or copy the definition of *settab2const* into your own script. If we make use of this function, fill the tables with data from Fig. 19.1, and “tweak” the tables to convert the entries to those given by Eqs. 19.4 and 19.5, the remainder of our function definition will look something like

```
call      {chanpath}      TABCREATE X 30 -0.100 0.050 // in volts
settab2const {chanpath} X_A 0 7 2.270
setfield {chanpath} X_A->table[8] 2.270 \ // -60 mV
X_A->table[9] 2.040 \ // -55 mV
X_A->table[10] 1.800 \ // -50 mV
.
.
X_A->table[29] 0.115 \ // 45 mV
X_A->table[30] 0.104 // 50 mV
settab2const {chanpath} X_B 0 8 0.0
```

```

setfield {chanpath} X_B->table[9] 0.0 \ // -55 mV
X_B->table[10] 0.011 \ // -50 mV
X_B->table[11] 0.0224\ // -45 mV
.
.
X_B->table[29] 0.550 \ // 45 mV
X_B->table[30] 0.585 // 50 mV
tweaktau {chanpath} X
end

```

The GENESIS Reference Manual describes another alternative to filling the tables with these many *setfield* commands. The data could also have been entered into a file and read into the **tabchannel** tables with the *file2tab* command.

There is one last trick that we can apply to wring the most efficiency out of our **tabchannel** simulations. If we were to set *xdivs* to 3000 instead of 30, and were to fill the tables with 3001 interpolated values, we could do without interpolation while the simulation is being run. Although this would result in a slight increase in the time required to set up the simulation, the execution would be a lot faster. The **tabchannel** has a TABFILL action that can be used to perform the table expansion and interpolation with the single command

```
call {chanpath} TABFILL X 3000 0
```

The last argument specifies that “fill mode” 0 (interpolation and smoothing with B-splines) will be used to fill the tables. Other options are to interpolate with cubic splines (mode 1) or to use linear interpolation (mode 2, the default). The *interp_struct* has a *calc_mode* field that can be set to determine whether or not interpolation will be used for a specified table. The following statement will set the mode to 0 (no interpolation) for the two tables

```
setfield {chanpath} X_A->calc_mode 0 X_B->calc_mode 0
```

These two statements should be added just before the final “end” statement in the function *make_B_trit_st*.

19.2.2 Testing and Editing the Channel

We should now test our channel model. The easiest way to do this is to make a simple cell containing this channel and use the *Neurokit edit channel* menu option to examine the channel properties. As we will only be looking at the channel itself and don’t care very much about the properties of the cell, we can modify any handy cell parameter file and *userprefs.g* file to construct the cell. An easy choice is to start with the *cell.p* file that was used in Chapter 17. The only thing we need to do to the *cell.p* file is to add the name of the channel *B_trit_st* and an arbitrary channel density to the line that specifies the contents

of the soma compartment. The *Na_squid.hh* and *K_squid.hh* channels can either remain or be deleted. Although this is not a reasonable thing to do in the context of this model, put a *B_trit_st* channel in the dendrite compartment, also. This will make it possible to experiment with separately modifying the behavior of the two B current channels. The only thing that needs to be done to the *userprefs.g* file is to add an *include* statement for the file containing the function *make_B_trit_st* and to invoke the function.

Once you have copied these files into the directory that contains your script for *make_B_trit_st* and have edited them to your satisfaction, load the model into *Neurokit*, following the procedure described in Section 17.4. Instead of choosing the *run cell* option from the title bar, click on *edit channel*. When the Cell Window appears in the lower left portion of the screen, click on the green spherical soma in order to select it. It should turn red, and icons representing the soma, the *B_trit_st* channel, and any other channels that are in the soma will appear in the Compartment Window at the upper left. Click on the *B_trit_st* channel in this window.

The Channel Parameters Window will appear in the lower right portion of the screen, below the Compartment Window. Figure 19.2 shows the group of dialog boxes, buttons, and toggles that are below the graphs in this window. The gate dialog box in this window will show the default gate to be examined, the X gate. As this is the one we want, position the cursor in this box and hit “Return”. If all has gone well, the two lower graphs will show plots for τ and m_∞ that agree with those in Figures 19.1A and 19.1B.

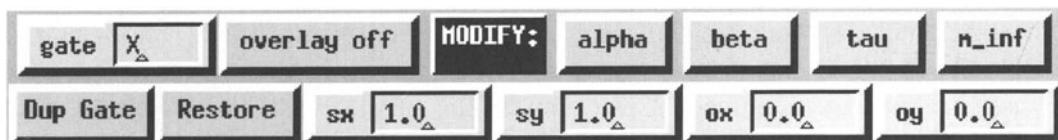


Figure 19.2 The control panel for the *Neurokit* “Channel Parameters Window”.

After making any corrections that are needed in order to reproduce the experimental data, we can try out some of the channel editing features. These will allow you to modify the characteristics of **tabchannels**, without having to edit the scripts that create them. The lower row of dialog boxes allows you to enter parameters for scaling or offset of any of the rate or state parameters.

We will begin by applying a vertical offset to uniformly increase the value of τ . Change the *oy* dialog box to read “0.5” and click on the button labeled *tau* to the right of the *MODIFY:* label. Notice that τ has been replotted and that the values have been shifted up by 0.5 sec.

Now, click on the dendrite compartment in the Cell Window. The Compartment Window should now show the dendrite, with the *B_trit_st* channel still selected. Hit “Return” in the gate dialog box, or click anywhere inside the box. Observe the resulting plot of τ for this gate. Apparently, the copy of the channel in the dendrite compartment has also been

modified. As they say in the software industry, this is a “feature,” not a “bug.” In a large compartmental model or large network, one may have many copies of a particular prototype channel. Usually, one wants these to behave identically. It is also desirable to minimize the amount of storage space used by the internal tables. For these reasons, copies of the channel that are created by the *copy* command or by *readcell* use the same tables as the original prototype, rather than new copies of the tables. This is true of all objects that contain tabular fields.

You may reverse this change by changing *oy* to -0.5 and clicking on *tau* again. Note that the scale and offset is always relative to the last operation, and not to the original values. Therefore, it is a good idea to set the dialog boxes back to offsets of 0.0 and scaling of 1.0 after making a change. This will prevent you from inadvertently making further changes.

Sometimes you may want to change just the one copy of the channel. GENESIS has a function that allows you to duplicate a **tabchannel** gate and its tables before modifying it. To try this, click on *Dup Gate* and repeat the experiment. You might also try using the *sy* to scale the values of τ , m_∞ , α , or β . To recover the old value, use the reciprocal of the previous scale factor.

m_∞ becomes non-zero shortly above -0.055 V . Suppose that we would like to give I_B a more rapid onset by shifting this point downward to -0.065 V . Start by clicking on *Dup Gate*, so that we won’t change the dendrite channel or the prototype in */library*. Then set the *ox* dialog to -0.01 and click on the *m_inf* button. After you have studied the results, reverse this change by using an offset of 0.01 . Then, repeat this a few times, shifting the curve to higher voltages each time you click on *m_inf*. Try restoring the original curve by changing *ox* to -0.01 and repeatedly clicking on *m_inf*. Do you see a problem?

One can reverse changes in *oy* and *sy*, because they just shift and scale the table values. However, *ox* and *sx* perform offsets and scaling of the horizontal axes by moving data in the tables. This can cause data to spill out of the ends of the tables and be lost. Thus, large changes in the *x*-axis should be avoided. Fortunately, there is a button labeled *Restore*. If you had the foresight to duplicate the gate, clicking on this button will restore the original from the prototype in */library*. If not, the prototype will have also been messed up and you will need to reload the cell. If you have the time, you might try another experiment with changing the *x*-axis. τ drops rapidly between -60 and -20 mV . Suppose that we would like to decrease the slope. We can do this by spreading the voltage scale out a bit. Experiment with the *sx* dialog to do this.

Most of these buttons and dialog boxes have a direct correspondence with GENESIS functions that you can use in your own simulation scripts. The scaling and offsets can be provided with the *scaletabchan* command. Bring the terminal window with the GENESIS prompt to the foreground and try the command “*scaletabchan -u*”. Like most commands that require additional arguments, this will result in a “usage” message. In this case, you should see

```
usage: scaletabchan channel-element gate mode sx sy ox oy [-duplicate]
Valid modes : a[lpha] b[eta] t[au] m[inf].
```

Now try

```
ce /cell/soma
scaletabchan B_trit_st X tau 1.0 1.0 0.0 0.5 -d
```

Put the terminal window into the background again so that you can see the Channel Parameters Window and click on the **gate** label of the dialog box that contains “X”. Notice that this produces the same results as if you had performed the vertical offset to τ by using the dialog boxes and **tau** button. After having “fine-tuned” your channel from within *Neurokit*, you may wish to permanently include these changes in your model. One way would be to edit your scripts that create the channels, changing the data that goes into the tables. An easier solution would be to use the original channel creation functions and to then use the *scaletabchan* function to perform the desired scaling and offset operations. If you are using scripts from the *Neurokit* prototypes library, or would like to use slightly different versions of the same channel in different simulations, it will be most convenient to use the *scaletabchan* function in your *userprefs.g* file, after you have created the prototype channels. Of course, you will probably not want to use the “*duplicate*” option of the function, as we did in the example above.

The **Dup Gate** button invokes the *duplicatable* command for both the internal tables of the specified channel and gate. If you like, you may try it out on one of the tables by giving the following commands.

```
pushe /cell/soma
duplicatable B_trit_st X_A
showfield B_trit_st X_A->table[900]
setfield B_trit_st X_A->table[900] 0.5555
showfield B_trit_st X_A->table[900]
pope /cell/dend
showfield B_trit_st X_A->table[900].
```

There is yet another way in which offsets and scaling may be provided to a **tabchannel** or **tabgate**. The **tabchannel** has fields *ox*, *oy*, *sx*, and *sy* for the *A* and *B* tables of each gate. These can be set with statements of the form

```
setfield B_trit_st X_A->ox 0.02.
```

This method is less convenient for modifying α , β , τ , or m_∞ , because it can only be used to change the scaling of the *A* and *B* tables. Note that changing *A* is not the same as changing α , because $B = \alpha + \beta$ will still use the old α value. However, this approach has

some advantages when the voltage scales for both τ and m_∞ are to be modified equally. Unlike the *scaletabchan* function, setting the *ox* and *sy* fields of a table simply changes the values of *xmin* and *xmax*, without moving data within the table. This procedure is not suitable for changing the scale of a single rate or state parameter, but can be used if both the *A* and *B* *x*-axes are modified equally. The *ox* field is particularly useful if one wants to adapt a channel for use in another cell that has a significantly different resting potential.

19.3 Using Equations for the Rate Constants

Often, you will be lucky enough to find a channel for which someone has already analyzed the experimental data and fitted the rate constants α and β to some function. This is the case for the channels that are used in the tutorial on the bursting hippocampal neuron in Chapter 7. The examples given here and in the next section are taken from the script *traub91chan.g* in the *neurokit/prototypes* directory. The functions that are defined in this script implement the channel models described in the paper by Traub et al. (1991).

The high voltage-activated calcium channel model used in this paper is typical of one that we might want to convert to a GENESIS **tabchannel**. In the notation of the paper, the current contributed by the channel is given by

$$I_{Ca} = \bar{g}_{Ca}s^2r(V - V_{Ca}). \quad (19.6)$$

The Hodgkin–Huxley rate constants that give rise to the activation variable s and the inactivation variable r have been fitted to the expressions

$$\alpha_s = \frac{1.6}{1 + \exp(-0.072(V - 65))} \quad (19.7)$$

$$\beta_s = \frac{0.02(V - 51.1)}{\exp\left(\frac{V-51.1}{5}\right) - 1} \quad (19.8)$$

$$\alpha_r = \begin{cases} 0.005 & [V \leq 0] \\ \frac{\exp(-V/20)}{200} & [V > 0] \end{cases} \quad (19.9)$$

$$\beta_r = \begin{cases} 0 & [V \leq 0] \\ 0.005 - \alpha_r & [V > 0]. \end{cases} \quad (19.10)$$

Although Eq. 19.6 uses an opposite convention for the direction of positive current flow than that used in Eq. 14.2, there is a straightforward correspondence between the variables used in Eq. 19.6 and the **hh_channel** and **tabchannel** field variables that are used in Eqs. 14.1 and 14.2. However, this paper uses the physiological units listed in Table 14.1, rather than SI units. With some care, we could also use physiological units in our channel model, but for consistency we will stick to SI units. In addition to converting the voltages from *mV* to

volts, we will have to scale the rate parameters by a factor of 1000, in order to make the conversion from inverse msec to inverse seconds. In the paper, all voltages are expressed with respect to a resting potential that is defined to be 0 mV, corresponding to an actual potential of -60 mV relative to the outside of the cell. We can make use of a GENESIS global variable, *EREST_ACT*, to give us some more flexibility in our function to create the channel. If we replace V by $V - EREST_ACT$ in Eqs. 19.7–19.10, we may use the channel with any assumed resting potential. In the script *traub91chan.g* and the example given below, *EREST_ACT* is set to -0.06 .

We would expect our function that creates this channel to use a *for* loop to fill each table. Because of the piece-wise continuous form of the inactivation rate constants, this will be necessary. However, GENESIS has a command, *setupalpha*, that can be used to create and fill the *A* and *B* tables when a gate has rate constants of the general form

$$y = \frac{A + Bx}{C + \exp((x + D)/F)}. \quad (19.11)$$

This is the case for Eqs. 19.7–19.8, so we will set up the tables for the *X* gate with a command of the form

```
setupalpha chan gate AA AB AC AD AF BA BB BC BD BF
```

where the parameters *AA*–*AF* correspond to those in Eq. 19.11 when it represents $\alpha(V)$ and the parameters *BA*–*BF* correspond to those for $\beta(V)$. The *setupalpha* function then performs many of the operations given in the example above for the *B_trit_st* channel. Namely, it

- calls the TABCREATE action for the specified gate (*X*, *Y*, or *Z*), setting *xdivs* to 49, *xmin* to -0.1 , and *xmax* to 0.05,
- loops over the *xdivs* + 1 table entries, using Eq. 19.11 to fill the *A* table with values for $\alpha(V)$ and the *B* table with values for $\beta(V)$,
- “tweaks” the *B* table, so that it contains $\alpha(V) + \beta(V)$,
- calls the TABFILL action to expand the table to 3000 entries, and sets the *calc_mode* field for each table to “no interpolation.”

Other optional parameters let you change the range and sizes given above.

Figure 19.3 shows a function that will create the channel *Ca_hip_traub91* using *setupalpha* to create the *X* (activation) gate. The *Y* (inactivation) gate and the *YA* and *YB* tables are created “the hard way” using the steps itemized above. As it is easy to go astray when converting between different units, you should make sure that you can reconcile Eqs. 19.7–19.8 with the values of the parameters used with *setupalpha*.

```
float EREST_ACT = -0.060 /* hippocampal cell resting potl */
float ECA = 0.140 + EREST_ACT // 0.080 volts
float SOMA_A = 3.320e-9      // soma area in square meters
function make_Ca_ hip_ traub91
    if ({exists Ca_ hip_ traub91})
        return
    end
    create tabchannel Ca_ hip_ traub91
    setfield
        ^          \
        Ek          {ECA}   \
                        // V
        Gbar        { 40 * SOMA_A } \
                        // S
        Xpower     2      \
        Ypower     1      \
        Zpower     0
    setupalpha Ca_ hip_ traub91 X 1.6e3 \
        0 1.0 {-1.0 * (0.065 + EREST_ACT) } -0.01389 \
        {-20e3 * (0.0511 + EREST_ACT) } \
        20e3 -1.0 {-1.0 * (0.0511 + EREST_ACT) } 5.0e-3
    float xmin = -0.1
    float xmax = 0.05
    int xdivs = 49
    call Ca_ hip_ traub91 TABCREATE Y {xdivs} {xmin} {xmax}
// Fill Y_A table with alpha and Y_B table with (alpha+beta)
    int i
    float x,dx,y
    dx = (xmax - xmin)/xdivs
    x = xmin
    for (i = 0 ; i <= {xdivs} ; i = i + 1)
        if (x > EREST_ACT)
            y = 5.0*{exp {-50*(x - EREST_ACT)}}
        else
            y = 5.0
        end
        setfield Ca_ hip_ traub91 Y_A->table[{i}] {y}
        setfield Ca_ hip_ traub91 Y_B->table[{i}] 5.0
        x = x + dx
    end
    setfield Ca_ hip_ traub91 Y_A->calc_mode 0    Y_B->calc_mode 0
    call Ca_ hip_ traub91 TABFILL Y 3000 0
end
```

Figure 19.3 An example script containing a function to create the high voltage-activated calcium channel.

19.4 Implementing Calcium-Dependent Conductances

Much of the interesting behavior of the bursting neurons discussed in Chapter 7 arises from potassium channels that have conductances depending on the intracellular concentration of calcium ions. In this section, we look at ways to obtain the calcium concentration from the calcium channel current, and examine the implementation of two types of calcium-dependent currents.

19.4.1 Calculating the Calcium Concentration

Several processes affect the concentration of calcium ions in a neural compartment (Yamada, Koch, and Adams 1989, Sala and Hernández-Cruz 1990). After calcium ions enter the compartment through calcium-selective ionic channels, they may diffuse into neighboring compartments. They may also bind to various buffers, such as the protein, calmodulin. Specialized ionic channels act as pumps to extrude calcium ions from the cell. GENESIS allows you to model one-dimensional diffusion of calcium with the **difshell** object. Non-mobile buffers can be simulated with the **fixbuffer**, and diffusible buffers with the **difbuffer** object. Two types of calcium pumps can be modeled, which can be made electrogenic. The **mmpump** object models a Ca-ATPase pump obeying Michaelis-Menten kinetics (Sec. 10.2.1), and the *setupNaCa* command uses a **tacurrent** object to model the $\text{Na}^+ - \text{Ca}^{2+}$ exchanger current. The **tacurrent** object allows you to model any non-ohmic currents, and it can also be used to compute the solution to the Goldman–Hodgkin–Katz equation, for which the appropriate tables are set up with the *setupghk* command. The mathematical equations implemented by these objects for calcium dynamics can be found in De Schutter and Smolen (1997). Examples of the use of these objects can be found in the spines tutorial (based on a model by Holmes and Levy (1990)) and the Purkinje cell tutorial. These, and other recently developed tutorials are available through the GENESIS Users Group (see Appendix A).

Often it is sufficient to use a simpler model, or there are not enough experimental data to adequately model the processes described above. The two simulations described in Chapter 7 represent the rate of change of the concentration of calcium ions by the equation

$$\frac{d[\text{Ca}^{2+}]}{dt} = BI_{\text{Ca}} - \frac{[\text{Ca}^{2+}]}{\tau}. \quad (19.12)$$

Here, $[\text{Ca}^{2+}]$ is the concentration, which we express in the SI units of moles per m^3 . This conveniently works out to be the same in milli-moles per liter. Note that the commonly used *Molar* concentration is expressed in moles per liter. Thus, the typical resting intracellular Ca^{2+} concentration of 50 nM would be 50×10^{-6} in our units. The first term on the right gives the rate of increase of $[\text{Ca}^{2+}]$ due to an inward channel current I_{Ca} . The second represents an attempt to fit the various processes that deplete $[\text{Ca}^{2+}]$ to an exponential decay

with a single time constant τ . An estimate of the value of τ may often be obtained by optical measurements of the rate of decay of $[Ca^{2+}]$, using Ca-sensitive dyes. In principle, one may find the constant B by calculating the flux of ions into a thin shell near the membrane surface produced by I_{Ca} , as in Exercise 3. This calculation leads to the result that for a shell of surface area A and thickness d ,

$$B = 5.2 \times 10^{-6} / (Ad), \quad (19.13)$$

where the shell dimensions are given in meters and the current is in amperes. However, there are factors that will modify this value, causing it to be just a rough approximation. Buffering ties up a large percentage of the entering Ca^{2+} , often reducing the effective value of B by a factor of 100 or more. The calcium concentration will not be uniform throughout the compartment, and the concentration in the shell close to the membrane surface is most relevant for behavior of channels that are activated or inactivated by the presence of calcium ions. As it is difficult to estimate an appropriate value to use for the thickness of the shell, d becomes yet another free parameter to be fitted. For the model of the bursting molluscan neuron of Chapter 7, the best available estimate of τ was used, and B was chosen so that Eq. 19.12 would yield maximum values of $[Ca^{2+}]$ which were in agreement with typical experimental measurements of the intracellular concentration of free Ca^{2+} .

GENESIS has an object, **Ca.concen**, which solves Eq. 19.12 for $[Ca^{2+}]$. There are data fields B , tau , and Ca for the corresponding quantities in Eq. 19.12. The calcium current appearing in the equation, I_{Ca} , is provided by summing current values that are provided by “I_Ca” messages from all channels which carry calcium currents. In addition, there is a $Ca.base$ field that is added to the resulting value of Ca in order to provide a “base” or resting value of the concentration.

In Chapter 16, Sec. 16.3 discusses the channel “density” parameter that is used in the cell descriptor file. When the “channel” that is specified in this file is the name of a prototype library element formed from a **Ca.concen** object, this value of the “density” is used to calculate the value of the B field, ignoring the value that was set when the prototype was created. This is analogous to the procedure used to set the $Gbar$ field of a **tabchannel**. For **Ca.concen** objects, B is set to the “density” divided by the volume of the parent compartment, with the volume calculated in m^3 from the dimensions given (in microns) in the cell descriptor file. By taking the actual compartment volume into account and setting an appropriate density, you can then set B for each compartment to be whatever you want. The object has an additional field $thick$, for the shell thickness d that appears in Eq. 19.13. If d is non-zero, the shell area A is calculated from the compartment dimensions, and B is set to the “density” divided by the volume Ad . Thus, the cell reader can also scale B as for a true shell.

The following statements define a function using the **Ca.concen** object to calculate the concentration of calcium ions resulting from a current flow through the *Ca.hip_traub91*

channel.

```

function make_Ca_hip_conc
    if ({exists Ca_hip_conc})
        return
    end
    create Ca_concen Ca_hip_conc
    setfield Ca_hip_conc \
        tau      0.01333  \      // sec
        B       17.402e12 \     // Curr to conc for soma
        Ca_base 0.0
    addfield Ca_hip_conc addmsg1
    setfield Ca_hip_conc \
        addmsg1      ".../Ca_hip_traub91 . I_Ca Ik"
end

```

In their model of the CA3 pyramidal cell, Traub et al. expressed $[Ca^{2+}]$ in arbitrary units with a resting concentration of 0, current in μA , and set $\tau = 13.33\ msec$. If we use the same concentration units, but express current in amperes and τ in seconds, our B constant is then 10^{12} times the constant (called ϕ) used in the paper. The actual value of B used for each compartment will typically be determined by the cell reader from the cell parameter file. However, for the prototype channel we will use Traub's value for the soma. (A misprint in the paper gives it as 17,402 rather than 17.402.) In our units, this is 17.402×10^{12} .

The *Ca_hip_conc* element created by this function should receive an “I_Ca” message from the calcium channel, accompanied by the value of the calcium channel current. If this message were stated explicitly in a simulation script, it would be

```
addmsg {path}/Ca_hip_traub91 {path}/Ca_hip_conc I_Ca Ik
```

Here, the variable *path* indicates the location of these two elements in the hierarchy. For example, *path* might evaluate to “/cell/soma”. However, we will ordinarily use the cell reader to create copies of these prototype elements in one or more compartments. We need some way to be sure that the needed messages are established. Although the cell reader has enough information to create the messages that link compartments to their channels and to other adjacent compartments, it must be provided with the information needed to establish additional messages. This is done by using the *addfield* command to place the message string in a user-defined field of one of the elements that is involved in the message. In our case, we use the statements

```

addfield Ca_hip_conc addmsg1
setfield Ca_hip_conc \
    addmsg1      ".../Ca_hip_traub91 . I_Ca Ik"

```

The cell reader recognizes the added fields *addmsg1*, *addmsg2*, etc. as indicating that they are to be evaluated and used to set up messages. The paths are relative to the element that contains the added message. Thus, “*../Ca_hip_traub91*” refers to the sibling element *Ca_hip_traub91* and “.” refers to the *Ca_hip_conc* element itself. Here we have chosen to attach the message to *Ca_hip_conc*. If we had attached it to *Ca_hip_traub91* instead, it would have been

```
addfield Ca_hip_traub91 addmsg1
setfield Ca_hip_traub91 \
    addmsg1      ".  ../Ca_hip_conc I_Ca Ik"
```

We will use the value of $[Ca^{2+}]$ produced by *Ca_hip_conc* to influence the conductance of two different types of potassium channels. Another use of the **Ca.concen** object would be to use the resulting concentration to change the value of an ionic equilibrium potential in situations where large changes in concentration can modify the driving force on the ions. The **nernst** object, which is described in the GENESIS Reference Manual, typically receives the intracellular ionic concentration with a CIN message and calculates the resulting equilibrium potential using the Nernst equation. The **tabchannel**, **tab2Dchannel** and **vdep_channel** objects (described below) can receive the new value of *Ek* from the **nernst** object with an EK message so that this field will be continually updated.

19.4.2 The AHP Current

The AHP current (Sec. 7.6) is a slow potassium current that depends only on $[Ca^{2+}]$. When time is expressed in seconds, the model used in the paper gives a rate constant α that increases linearly from 0 to 10 as $[Ca^{2+}]$ increases from 0 to 500. When $[Ca^{2+}]$ is greater than 500, in these arbitrary units, α has a constant value of 10. The β parameter has a constant value of 1.0 over the entire range of $[Ca^{2+}]$.

This concentration-dependent activation can be modeled with the **tabchannel** *Z* gate. The *Z* gate acts just like the *X* and *Y* gates, except that it gets its input value from a CONCEN message, instead of a VOLTAGE message. The parameter that is sent is usually an ionic concentration, coming from a **Ca.concen** object. (There is no reason that this message couldn't be used to send a voltage or other variable, however.)

The *traub91chan.g* script defines a function to create the channel *Kahp_hip_traub91*. This is very similar to the function shown in Fig. 19.3, which creates the *Ca_hip_traub91* channel. However, the exponents *Xpower* and *Ypower* are set to zero, and *Zpower* is set to 1. A *for* loop is used to fill the *A* and *B* tables for the *Z* gate and the table is expanded with TABFILL, as before. In addition, the cell reader needs to be given the information necessary for it to set up a CONCEN message from the *Ca_hip_conc* element. This is accomplished with the statement

```

addfield Kahp_hip_traub91 addmsg1
setfield Kahp_hip_traub91 \
    addmsg1          ".../Ca_ hip_conc . CONCEN Ca"

```

As with the added field that was defined for the *Ca_ hip_conc* element, the string assigned as the value of *addmsg1* is used only by the cell reader.

19.4.3 The C-Current

The *Z* gate can be used only if the conductance has a factor Z^{Zpower} , where *Z* obeys Hodgkin–Huxley rate equations like those for *X* and *Y*, but with α and β being functions of concentration. However, the concentration dependence is not always of this form. The C-current (Chapter 7) is a fast potassium current that depends on both voltage and the calcium concentration. The model used by Traub, et al. (1991) for the conductance has a typical voltage and time dependent activation gate, where the time dependence arises from the solution of a differential equation containing the rate constants α and β . It is multiplied by a function of calcium concentration that is given explicitly, rather than being obtained from a differential equation. Therefore, we need a way to multiply the activation by a concentration-dependent value that is determined from a lookup table.

GENESIS doesn't have a way to implement this with a **tabchannel**, so we use the **vdep_channel** object here. These channels contain no gates and get their activation gate values from external gate elements, via a MULTGATE message. These gates are usually created with **tabgate** objects, which are similar to the internal gates of the **tabchannels**. However, any object that can send the value of one of its fields to the **vdep_channel** can be used as the gate. Here, we use the **table** object. This generality makes the **vdep_channel** very useful, but it is slower than the **tabchannel** because of the extra message passing involved. The following function illustrates the steps needed to implement this channel.

```

float EREST_ACT = -0.060 /* hippocampal cell resting potl */
float EK = -0.015 + EREST_ACT // -0.075
float SOMA_A = 3.320e-9      // soma area in square meters
function make_Kc_hip_traub91
    if ({exists Kc_hip_traub91})
        return
    end
    create vdep_channel   Kc_hip_traub91
    setfield           ^          \
                    Ek          {EK}          \          //          V
                    gbar         { 100.0 * SOMA_A }          //          S
    float   xmin = 0.0
    float   xmax = 1000.0
    int     xdivs = 50

```

```
create table          Kc_hip_traub91/tabc
call Kc_hip_traub91/tabc TABCREATE {xdivs} {xmin} {xmax}
int i
float x,dx,y
dx = (xmax - xmin)/xdivs
x = xmin
for (i = 0 ; i <= {xdivs} ; i = i + 1)
    if (x < 250.0)
        y = x/250.0
    else
        y = 1.0
    end
    setfield Kc_hip_traub91/tabc table[{i}] {y}
    x = x + dx
end
setfield Kc_hip_traub91/tabc table->calc_mode 0
call Kc_hip_traub91/tabc TABFILL 3000 0
// Now make a tabgate for the voltage-dependent activation parameter.
float xmin = -0.1
float xmax = 0.05
int xdivs = 49
create tabgate      Kc_hip_traub91/X
call Kc_hip_traub91/X TABCREATE alpha {xdivs} {xmin} {xmax}
call Kc_hip_traub91/X TABCREATE beta  {xdivs} {xmin} {xmax}
int i
float x,dx,alpha,beta
dx = (xmax - xmin)/xdivs
x = xmin
for (i = 0 ; i <= {xdivs} ; i = i + 1)
    if (x < EREST_ACT + 0.05)
        alpha = {exp {53.872*(x - EREST_ACT) - 0.66835}}/0.018975
        beta = 2000*{exp {(EREST_ACT + 0.0065 - x)/0.027}} - alpha
    else
        alpha = 2000*{exp {-(EREST_ACT + 0.0065 - x)/0.027}}
        beta = 0.0
    end
    setfield Kc_hip_traub91/X alpha->table[{i}] {alpha}
    setfield Kc_hip_traub91/X beta->table[{i}] {beta}
    x = x + dx
end
setfield Kc_hip_traub91/X alpha->calc_mode 0 beta->calc_mode 0
call Kc_hip_traub91/X TABFILL alpha 3000 0
call Kc_hip_traub91/X TABFILL beta  3000 0
addmsg Kc_hip_traub91/tabc Kc_hip_traub91 MULTGATE output 1
addmsg Kc_hip_traub91/X  Kc_hip_traub91  MULTGATE m 1
```

```

addfield Kc_hip_traub91 addmsg1
addfield Kc_hip_traub91 addmsg2
setfield Kc_hip_traub91 \
    addmsg1      ".../Ca_hip_conc tab INPUT Ca" \
    addmsg2      "... X VOLTAGE Vm"
end

```

First, we create the **vdep_channel**, *Kca_hip_traub91*, and set the usual fields. Then, we create the lookup table for the function of concentration, $f([Ca^{2+}]) = \min(1, [Ca^{2+}] / 250)$. It is made from a **table** object, and filled in a manner similar to that used for the internal tables of the **tabchannel** object. Note that the internal field for the table is called *table*. When expanding the table so that it may be used in the “no interpolation” calculation mode, note that the TABFILL syntax is slightly different from that used with **tabchannels**. Here, there is only one internal table, so the table name is not specified.

Next, we make a **tabgate** for the voltage-dependent rate constant for activation. The **tabgate** has two internal tables, *alpha* and *beta*. These are filled like those of the **tabchannel**. Note that the *beta* table is really β , not $\alpha + \beta$, as with the *B* table of the **tabchannel**. Finally, we set up the required messages. The MULTGATE message is used to give the **vdep_channel** the value of the activation variable and the power to which it should be raised. As we have created the **tabgate** and **table** as subelements of the channel, they and their messages to the channel will accompany it when copies are made. However, we also need to provide for messages that link to external elements. The message that sends the Ca^{2+} concentration to the **table** and the one that sends the compartment membrane potential to the **tabgate** are stored in added fields of the channel, so that they may be found by the cell reader.

Later in this chapter, we will see how the **tab2Dchannel** allows the efficient modeling of more complex relationships between the channel conductance and V_m and $[Ca^{2+}]$, and how future versions of the **tab2Dchannel** might be used to implement this channel model without the use of the **vdep_channel**.

19.4.4 Other Uses of the **table** Object

The **table** object is quite versatile. There is also a similar **table2D** object that has an internal two-dimensional table. These may both be used in many situations where there is no GENESIS object that will perform a given calculation, or as a faster alternative to another object. For example, Section 19.4.1 mentioned the use of the **nernst** object to continually update the equilibrium potential of a **tabchannel**. The script *mitproto.g* in the *neurokit/prototypes* directory shows how this may also be accomplished with a **table**. The *Cable* tutorial uses this object as an intermediary to the **xgraph** object in order to generate logarithmic plots. In Chapter 22 (Sec. 22.4.2) we will see an example of its use as a function generator. Another use of the **table** would be to implement an “instantaneous” gate that has an activation depending only on voltage, rather than having a time and voltage dependence

given by the solution of Eq. 19.3. You can find further examples in the *Scripts/examples/table* directory.

19.4.5 The **vdep_gate** Object

For completeness, we should mention the **vdep_gate** object, which may also be used with a **vdep_channel**. This gate is very much like the **tabgate**, except that it calculates the rate constants α and β directly from Eq. 19.11, instead of from internal tables. Although this form is slightly more general than the three forms used by the **hh_channel**, an **hh_channel** implementation will execute somewhat faster than the **vdep_channel** and **vdep_gate** combination. If it is necessary to use separate channel and gate objects, as with the C-current, a combination of a **vdep_channel** and **tabgate** will give the most speed and flexibility. Thus, there is little reason to use the **vdep_gate**. It is seen mainly in older GENESIS simulations that were written before the development of the **tabchannel** and **tabgate** objects.

19.4.6 Using the **tab2Dchannel** Object

Some channel models require rate constants that depend on another variable in addition to V_m . For example, Moczydlowski and Latorre (1983) have described the kinetics of a calcium-activated potassium channel that has been widely used as a model for the C-current. Unlike the current described in Sec. 19.4.3, the dependence of the conductance on $[Ca^{2+}]$ arises not from a multiplicative function of $[Ca^{2+}]$, but from the dependence of α and β on both V_m and $[Ca^{2+}]$.

In GENESIS, such models may be implemented with the **tab2Dchannel**, which contains two-dimensional tables for the variables A and B . Like the **tabchannel**, and unlike the **vdep_channel** or **hh_channel**, the **tab2Dchannel** may be used with the fast and highly accurate implicit numerical integration methods that are described in Chapter 20. The listing of the script *MoczydKC.g* in Fig. 19.4 reveals some of the significant features of the **tab2Dchannel**, and allows us to compare it with the **tabchannel**.

In comparing this listing with that in Fig. 19.3, we see that most of the channel fields are the same. However, the TABCREATE action now takes additional arguments *ydivs*, *ymin* and *ymax* in order to allocate the two-dimensional tables for A and B . The tables now have two indices, where the first one runs from 0 to *xdivs* and the second one from 0 to *ydivs*.

When using two-dimensional tables, it may be necessary to experiment with the table size in order to obtain the desired accuracy without using an excessively large table. The setup time needed to fill the tables using a *for* loop is also a consideration, as there are presently no utilities like the *setuplapha* command for filling **tab2Dchannel** tables. (But, check the GENESIS Reference Manual for new developments.)

When using a one-dimensional table, it is customary to use a large table, either by setting *xdivs* to a large value, or by using TABFILL to expand the table with interpolated

```

/* non-inactivating BK-type Ca-dependent K current
** Moczydlowski and Latorre 1983, J. Gen. Physiol. 82:511-542.
** Implemented by Erik De Schutter BBF-UIA,
** with original parameters scaled for units: V, sec, mM */
float EK = -0.085 // volts
function make_Moczyd_KC
    int xdivs = 100
    int ydivs = {xdivs}
    float xmin, xmax, ymin, ymax
    xmin = -0.1; xmax = 0.05; ymin = 0.0; ymax = 0.0030
    int i, j
    float x, dx, y, dy, a, b
    float Temp = 37
    float ZFbyRT = 23210/(273.15 + Temp)
    if (!({exists Moczyd_KC}))
        create tab2Dchannel Moczyd_KC
        setfield Moczyd_KC Ek {EK} Gbar 0.0 \
            Xindex {VOLT_C1_INDEX} Xpower 1 Ypower 0 Zpower 0
        call Moczyd_KC TABCREATE X {xdivs} {xmin} {xmax} \
            {ydivs} {ymin} {ymax}
    end
    dx = (xmax - xmin)/xdivs
    dy = (ymax - ymin)/ydivs
    x = xmin
    for (i = 0; i <= xdivs; i = i + 1)
        y = ymin
        for (j = 0; j <= ydivs; j = j + 1)
            a = 480*y/(y + 0.180*{exp {-0.84*ZFbyRT*x}}})
            b = 280/(1 + (y/(0.011*{exp {-1.00*ZFbyRT*x}}))))
            setfield Moczyd_KC X_A->table[{i}][{j}] {a}
            setfield Moczyd_KC X_B->table[{i}][{j}] {a + b}
            y = y + dy
        end
        x = x + dx
    end
    setfield Moczyd_KC X_A->calc_mode {LIN_INTERP}
    setfield Moczyd_KC X_B->calc_mode {LIN_INTERP}
    addfield Moczyd_KC addmsg1
    setfield Moczyd_KC addmsg1 ".../Ca_conc . CONCEN1 Ca"
end

```

Figure 19.4 An example script using a **tab2Dchannel** to implement the Moczydlowski and Latorre (1983) Ca-dependent K current.

values. Then, as was done in Fig. 19.3, the *calc_mode* field for each table is set to zero (*NO_INTERP*), in order to save computation time. In the present example, the *calc_mode* is set to *LIN_INTERP* (a predefined global variable equal to one), so that linear interpolation is performed at run time, allowing the use of a smaller table.

This script uses a maximum value of $[Ca^{2+}]$ (*y_{max}*) of 0.003 mM (milli-moles per liter), because it was intended to be used with a Ca channel and associated **Ca_concen** element that produced concentrations in this range. The channel parameters were fitted to experimental data for a range of concentrations up to 10 mM, so your value of *y_{max}* could be much larger. As always, it is important to take some care in choosing the value of the parameter *B* in Eq. 19.12 and the corresponding field in the **Ca_concen** element, and then determine the maximum value of $[Ca^{2+}]$ that the **Ca_concen** element will generate for your particular model.

As with the **tabchannel**, a message carrying the membrane voltage or a concentration is sent to the channel so that the channel can retrieve the appropriate *A* and *B* table values to calculate the gate activations (*X*, *Y* and *Z*) and the resulting channel conductance. However, we can now have two messages, in order to specify both the *x* and *y* variables. There are two new messages for sending concentrations (or anything else), **CONCEN1** and **CONCEN2**. Another message, **DOMAINCONC**, provides a highly simplified model to obtain the ionic concentration directly, using the current sent from another channel and the surface area of the parent compartment (De Schutter and Smolen 1997). There are also three new fields *Xindex*, *Yindex* and *Zindex*. These fields are used for each gate to define which message refers to the *x* variable and which refers to the *y* variable. These index fields may each be assigned to one of the predefined global variables **VOLT_INDEX**, **C1_INDEX**, **C2_INDEX**, **DOMAIN_INDEX**, **VOLT_C1_INDEX**, **VOLT_C2_INDEX**, **VOLT_DOMAIN_INDEX**, **C1_C2_INDEX** and **DOMAIN_C2_INDEX**.

The first four of these are used when a gate depends on only one variable. In this case, *xdivs* should be set to zero for that gate, and the *y* variable (corresponding to the second index) is used to fill the *A* and *B* tables. Then, the prefix (VOLT, C1, C2 or DOMAIN) specifies whether the VOLTAGE, CONCEN1, CONCEN2, or DOMAINCONC message is used to provide the *y* variable. The remaining five of these variables are of the form *x_y_INDEX*, and similarly specify which of two messages are used to specify the *x* and *y* variables.

In our example, *Xindex* is set to **VOLT_C1_INDEX**. This means that a VOLTAGE message will specify the *x* variable of the *X_A* and *X_B* tables, and a CONCEN1 message will specify the *y* variable. The cell reader will automatically provide the VOLTAGE message from the parent compartment. However, as with the C-current model described in Sec. 19.4.3, we need to use an added field *addmsg1* to hold a string that tells the cell reader to create a CONCEN1 message from a sibling **Ca_concen** element (which we have called *Ca_conc*) to our channel, giving the Ca concentration. If our model required a second gate *Y* that depended solely on another ionic concentration, we could also send a CONCEN2 message,

and set *Yindex* to *C2_INDEX*.

Future versions of the **tab2Dchannel** will allow the *Xindex*, *Yindex* and *Zindex* fields to take on values to indicate that the corresponding gate should be given a value instantaneously calculated from the *A* table values, rather than from the solution of Eq. 19.3. This will allow the fast and efficient modeling of “instantaneous” channels and C-current models like those described in Sec. 19.4.3 without the use of **table** or **vdep_channel** objects.

19.5 NMDA Channels

Postsynaptic receptors that are activated by N-methyl-D-aspartate (NMDA) have characteristics that differ significantly from other receptors that give rise to EPSPs. NMDA channel conductances have rise times on the order of 5–10 msec and decay slowly over periods of 70–100 msec. In addition to showing this prolonged conductance, NMDA channels have a voltage-dependent conductance that increases with depolarization from the resting potential. As a significant synaptic current requires both a presynaptic input and a postsynaptic depolarization, NMDA synapses can act like Hebbian synapses, showing a use-dependent facilitation. NMDA channels pass a combination of sodium, potassium and calcium ions. Experiments have shown that mechanisms for long term potentiation (Sec. 15.4) depend upon a postsynaptic increase in the intracellular Ca^{2+} concentration. This increase is thought to be brought about by calcium influx through NMDA channels. For these reasons, there has been a great deal of interest in the modeling of NMDA channels.

The voltage dependence of NMDA channels is caused by a blockage of the channel by magnesium ions. At membrane potentials on the order of -70 mV , the driving force for these large ions to try to enter the channels is quite high. As the membrane is depolarized, the blockage is relieved. The behavior of the magnesium block has been modeled by Jahr and Stevens (1990). Zador, Koch and Brown (1990) have used this model and data from hippocampal neurons to fit the NMDA channel conductance to

$$g_{syn}(V, t) = g_n \frac{\exp(-t/\tau_1) - \exp(-t/\tau_2)}{1 + \eta[Mg^{2+}] \exp(-\gamma V)}. \quad (19.14)$$

In this expression, the constants $g_n = 0.2\text{ nS}$, $\tau_1 = 80\text{ msec}$, $\tau_2 = 0.67\text{ msec}$, $\eta = 0.33/\text{mM}$, and $\gamma = 0.06/\text{mV}$. Typical values of extracellular $[Mg^{2+}]$ in hippocampal slice experiments are about 2 mM . The ratio of sodium, potassium and calcium ions passed by this channel is such that the reversal potential is close to the resting potential.

The time dependence of Eq. 19.14 is the same as that of the dual exponential form of the **synchan** object (Eqs. 6.17 and 15.2). In GENESIS, we can implement NMDA channels by multiplying the conductance of a **synchan** by a voltage-dependent term which is provided by another object, the **Mg_block**. This object has a field *CMg* that corresponds to $[Mg^{2+}]$, a

field KMg_A corresponding to $1/\eta$, and a field KMg_B corresponding to $1/\gamma$. Note that any units may be used for the concentrations CMg and KMg_A , as long as they are consistent.

The following statements illustrate the use of a **synchan** and **Mg_block** object to implement an NMDA channel.

```

float CMg = 2                                // [Mg] in mM
float eta = 0.33                             // per mM
float gamma = 60                            // per Volt
create    synchan      {compartment}/{channel}
setfield   ^ \
            Ek          {Ek} \
            tau1        {tau1} \
            tau2        {tau2} \
            gmax        {gmax}
create Mg_block {compartment}/{channel}/block
setfield   ^ \
            CMg         {CMg} \
            KMg_A       {1.0/eta} \
            KMg_B       {1.0/gamma}
addmsg   {compartment}/{channel} {compartment}/{channel}/block \
         CHANNEL Gk Ek
addmsg   {compartment}/{channel}/block {compartment} CHANNEL Gk Ek
addmsg   {compartment}   {compartment}/{channel}/block VOLTAGE Vm
// Even though we don't use the channel current, CHECK expects this message
addmsg   {compartment}   {compartment}/{channel} VOLTAGE Vm

```

After creating the two elements and setting the fields for the various constants that appear in Eq. 19.14, we set up a CHANNEL message to pass the unmodified conductance from the **synchan** to the **Mg_block**. Another CHANNEL message passes the blocked conductance to the parent compartment from the **Mg_block**. The **Mg_block** needs the compartment membrane potential to calculate the voltage-dependent factor in Eq. 19.14, so it gets it via a VOLTAGE message. The **synchan** expects to receive a VOLTAGE message in order to calculate a channel current. Although the value of the current in the absence of blocking is of no interest to us, we send the message anyway, in order to avoid error messages produced by the CHECK action of the channel.

19.6 Gap Junctions

In addition to communicating via chemically activated synapses, neurons may communicate through direct electrical connections. These *gap junctions*, which are common in lower vertebrates and found in several sites of the mammalian brain, are large macromolecules that extend through the membranes of both cells. Pores in these molecules allow the rapid

exchange of ions between the two neurons without the delays that occur with synaptic coupling. As they offer a very low resistance with little leakage to the extracellular space, the postsynaptic potential is nearly the same as the presynaptic potential.

The speed of this type of coupling makes it useful for modifying the behavior of networks of coupled oscillators. Studies of the pyloric network of the lobster stomatogastric ganglion have shown that the frequency of the anterior burster neuron is influenced by electrical coupling to other neurons (Hooper and Marder 1987).

Neurons in the inferior olfactory nucleus are extensively connected with gap junctions, most of which seem to exist between the spines that populate the dendritic trees of these cells. These have been simulated in GENESIS by using an RAXIAL message passed directly from each dendritic compartment to the other, with the *Ra* field replaced by a number representing the resistance of the gap junction.

A typical specific membrane resistance for a gap junction is on the order of $10^{-4} \Omega m^2$, which is much lower than that for a typical cell membrane. For two compartments that are coupled by a junction with a $1 \mu m$ diameter, the resistance of the coupling would be about $30 M\Omega$. In such a case, the following messages might be used to couple two cells with a gap junction.

```
float Rgap = 3e7
addmsg /cell1/dendrite /cell2/dendrite RAXIAL {Rgap} Vm
addmsg /cell2/dendrite /cell1/dendrite RAXIAL {Rgap} Vm
```

If the value of *Rgap* is changed, these messages must be deleted and re-sent with the new value.

19.7 Dendrodendritic Synapses

Mitral cells in the olfactory bulb interact with axonless granule cells via dendrodendritic synapses (Rall and Shepherd 1968, Shepherd and Greer 1990). In GENESIS, these synapses may be modeled either by a combination of a **synchan** and a **table** object, or by the **ddsyn** object. Both approaches are illustrated by a demonstration in the *Scripts/examples/ddsyn* directory.

The **ddsyn** object is much like a **synchan**, with the same internal fields and messages. However, it also has an internal table that maps a presynaptic potential to the channel activation. In a typical usage, it gets its activation, not in the form of a δ -function spike from an axonal connection, but from its internal table in response to a voltage sent from the presynaptic dendrite compartment with a V_PRESYN message. This activation is then used to generate a channel conductance governed by two time constants, τ_1 and τ_2 , as with the **synchan**.

19.8 Exercises

1. Convert one of the channels in *hhchan.g* into a **tabchannel**. Test your channel and verify that it gives the same results as the original.
2. Yamada, Koch and Adams (1989) give a slightly different model for the AHP current from the one used in the Traub model. In MKS units, they use

$$\alpha(C) = f(C)$$

$$\beta = 2.5(1/\text{sec}) = \text{const}$$

$$f(C) = 1.25 \times 10^8 C^2,$$

with $C = [\text{Ca}^{2+}]$ in *mmoles/liter* = *moles/m*³. Use the **tabchannel** *Z* gate to implement a model of this current. Incorporate it into a simple cell and describe its effects.

3. The term BI_{Ca} in Eq. 19.12 represents the number of moles of Ca²⁺ ions that enter a unit volume each second, due to a current I_{Ca} . Assume that the volume is a thin shell with a surface area A and a thickness d . Show that when SI units are used, $B = 5.2 \times 10^{-6}/Ad$.
4. Modify the script from Chapter 15 so that it uses an NMDA channel. Use the parameters values given with Eq. 19.14, but expressed in SI units. Choose a value of *gmax* that results in occasional action potentials, but mostly subthreshold EPSPs. Also modify the graph for the conductance plot so that it plots both the unblocked conductance of the **synchan** element and the net conductance of the **Mg.block** element. Compare these two plots and the plot of the soma membrane potential. Can you verify that the NMDA channel is behaving as one would expect during the course of an action potential?
5. The original Traub et al. (1991) CA3 pyramidal cell model had a quisqualate-activated conductance and an NMDA conductance that were not implemented in the model used in Chapter 7. From the description given in their paper, implement these conductances and use *Neurokit* to provide them with synaptic input. Verify that the model behaves as expected.

Chapter 20

Speeding Up GENESIS Simulations

ERIK DE SCHUTTER and DAVID BEEMAN

20.1 Introduction

As your GENESIS simulations grow in complexity, either through increased level of detail in multi-compartmental cell models, or through increased size of network models, you will eventually begin to look for ways to increase the speed of your simulations. In the following section, we present some general hints, methods and “tricks” that you can use to make your simulations run faster. For models that contain many compartments, you can obtain large speedups by using a more efficient numerical method for the integration of the compartmental equations. In fact, as we will see in Sec. 20.3.4, the default integration method that is used in GENESIS is much better suited to simple cell models with few compartments. The rest of this chapter describes how you may implement these improved methods, with emphasis on the fast *implicit* methods that are associated with the GENESIS **hsolve** object.

20.2 Some General Hints

Here are some general suggestions for speeding up a GENESIS simulation:

Use multiple clocks A typical neural simulation has a variety of elements whose behavior spans a range of time scales. As we have seen in Chapter 7, action potentials produced by sodium channels have very rapid rise times, whereas some varieties of potassium channels and changes in calcium concentration have time scales that can be on the

order of minutes. Many simulators use a single integration step size for all parts of the simulation and often automatically choose the integration step that is to be used. Although this is a convenience to the user, it gives you less control over factors that affect the speed and accuracy of the simulation. Chapter 13 described the use of `useclock` and `setclock` for assigning different clocking rates to the simulation of different elements. You can greatly speed up a simulation by using slower clocks for simulation elements that can be updated at a slower rate. This is particularly useful in the case of graphical elements. Some loss in resolution in plotted results can greatly increase the speed of the simulation with no cost to the overall accuracy of the results. It is not advisable to run computational elements (**channels**, **compartments**, ...) at different clock speeds, as this will often reduce the simulation accuracy to that of the element with the slowest clock. In case of doubt, you should make sure that your simulation is still giving the same results after making any changes to clocking rates.

Perform runs in “batch” mode Graphical output is very slow, because many plot messages are being exchanged at every simulation step. Most large research simulations perform very little graphical output. The important data are written to a file to be plotted and analyzed later. After you have tested your simulation interactively, modify the scripts to allow it to run without user input or graphical output when you are ready for long runs. The *Piriform* and *MultiCell* simulation scripts give examples of how you may cleanly separate the graphical and non-graphical simulation code. This is also discussed in Sec. 22.7. The `save` and `restore` commands (described in the GENESIS Reference Manual) can be used if you need to split the simulation into a number of runs that begin at the state where the previous one stopped.

Simplify cell models Using the simplest possible cell model that has the desired characteristics can make your network simulations run faster. For example, the somata in the original Wilson and Bower (1989) olfactory cortex model had no Hodgkin–Huxley Na and K channels. Instead of generating true action potentials, a soma simply passed its V_m to a spike generator that fired a spike when V_m crossed a threshold. Often one does some short runs with a fairly detailed model, and then starts making simplifications and approximations until an acceptable simple model is found. The *Piriform* simulation scripts include the capability of using either this *integrate-and-fire* model or the Hodgkin–Huxley model for action potential generation.

Use table lookup Many calculations can be greatly speeded up by looking up numerical values in a table, rather than calculating them. This is why **tabchannel** and **tab2Dchannel** elements are processed much faster than **hh_channel** elements. Section 19.4.4 mentions ways to use the **table** object to replace other objects that normally perform a calculation.

Use compiled functions Any GENESIS script that involves a large number of steps in a loop is likely to be slow, because it is being processed by an interpreter. That is why most GENESIS scripts merely create instances of precompiled objects and set up messages to be passed between them. All of the real computation is done with compiled code that is either part of compiled function definitions or part of the actions defined for an object. If you find that you are using script language functions in a manner that causes them to be invoked repeatedly during the course of a simulation, it may be a good idea to write new functions in C, or write C code to define new actions for an object. These may then be compiled into your own version of GENESIS, as described in the GENESIS Reference Manual section on Customizing GENESIS.

Reduce “swapping” to disk If the progress of your simulation slows down significantly and you begin hearing the sounds of increased disk activity, this is a sign that your simulation has used up the amount of conventional memory that is available and is swapping out parts of your simulation to disk. The GENESIS *showstat -process* command can be used to tell how much memory your simulation is using at any point. If closing down other applications that are running at the same time doesn’t give you enough memory, it may be worth it to install more memory in your machine, or to find ways to reduce the memory requirements of your simulation.

Use parallel computation GENESIS has been ported successfully to several parallel computer platforms and to networked workstations. Both networks and single neuron models have been parallelized by researchers using GENESIS. A version of GENESIS, called PGENESIS, is available for use with the Parallel Virtual Machine (PVM), allowing GENESIS to run on the many systems that support the PVM standard. Using parallel computers has several potential advantages. First, they can increase execution speed, but there is a trade-off as increasing the number of computing processors also increases the communication overhead between these processors. Second, because the model is distributed over a lot of processors’ memory, huge models can be implemented. The simplest way to use parallel computers is as a “CPU-farm,” without any communication between nodes. This is useful in situations where you need to run the same simulation many times, using different sets of parameters, in order to perform parameter space searches to “fine-tune” the model (Bhalla and Bower 1993, De Schutter and Bower 1994c, Vanier and Bower 1996). This does not require any special code and is equivalent to running GENESIS on a lot of workstations.

To get true parallelism, PGENESIS defines a new object, called the **postmaster**, which takes care of all communication between processors (either on a parallel computer or between different workstations). Using the **postmaster** is rather transparent. For example, one can send a message to an element on another processor. For more information about using PGENESIS, see Chapter 21 and the PGENESIS documentation.

Use faster numerical methods We have used the default integration method in the simulations that we have constructed so far. However, you may have noticed that the *Neurokit* implementation of the *Burster* tutorial of Chapter 7 has a method dialog box showing “11” and that the *Cable* tutorial of Chapter 5 provides a menu for selecting the numerical method to be used. As we mentioned earlier (Sec. 2.4.2), there are tradeoffs between speed, accuracy, and flexibility when choosing a numerical method. The remainder of this chapter describes how you may use much faster numerical methods in your simulations, and how to deal with some of the restrictions that they place on your ability to modify the simulations.

20.3 Numerical Methods Used in GENESIS

In order to explain some of the issues involved in choosing an appropriate numerical integration method, we present here a discussion of the use of numerical methods for solving the equations that arise in neural simulations. You may skip the rest of this section if you are not interested in this level of detail. If you are interested in learning more about this subject, you may wish to read the review by Mascagni (1989).

20.3.1 The Differential Equations Used in GENESIS

Equation 2.1, giving the membrane potential in a generalized neural compartment, and Eq. 4.4 for the state of a Hodgkin–Huxley channel gate, are typical of the differential equations to be solved in a neural simulation. Both of these are members of a set of N coupled first-order ordinary differential equations of the general form

$$\frac{dy_i}{dt} = f(t, y_1, y_2, \dots, y_N), \text{ with } i = 1, \dots, N. \quad (20.1)$$

In order to simplify the notation, we drop the subscript i and write the general equation to be solved as

$$\frac{dy}{dt} = f(t). \quad (20.2)$$

However, you should keep in mind that the dependence on the time t usually enters implicitly through the time dependence of the various y ’s. For example, in Eq. 2.1, the membrane potentials in the adjacent compartments V_m'' and V_m' , the conductances G_k , and the injection current I_{inject} all depend on t .

20.3.2 Explicit Methods

Forward Euler Method

The *forward Euler* method is the simplest of the numerical methods available to GENESIS for the solution of Eq. 20.2. For a time increment Δt , we approximate $y(t + \Delta t)$ by

$$y(t + \Delta t) = y(t) + f(t)\Delta t. \quad (20.3)$$

This approximation is equivalent to keeping only the first derivative in a Taylor series expansion,

$$y(t + \Delta t) = y(t) + \frac{dy}{dt}\Delta t + \frac{1}{2} \frac{d^2y}{dt^2}(\Delta t)^2 + \frac{1}{6} \frac{d^3y}{dt^3}(\Delta t)^3 + \dots \quad (20.4)$$

These terms that involve the higher derivatives can often be very large. We will later see that the forward Euler method suffers from instability problems. Thus, this method is a relatively poor approximation to the solution, and is rarely used in GENESIS simulations.

Adams-Basforth Methods

The *Adams-Basforth* methods approximate these missing higher derivatives by making use of past values of $f(t)$ in the approximation for $y(t + \Delta t)$. The general form is

$$y(t + \Delta t) = y(t) + \Delta t(a_0 f(t) + a_1 f(t - \Delta t) + a_2 f(t - 2\Delta t) + \dots + a_n f(t - n\Delta t)), \quad (20.5)$$

where the coefficients a_n may be found by expanding $f(t - n\Delta t)$ in a Taylor's series and comparing Eq. 20.5 with Eq. 20.4. If we evaluate f at n previous times in Eq. 20.5, we say that this is an $(n + 1)th$ order Adams-Basforth method, because it corresponds to keeping terms through the $(n + 1)th$ derivative in the Taylor's series expansion. GENESIS lets you choose between second- through fifth-order Adams-Basforth methods. These methods are computationally very efficient, as they achieve higher accuracy by making use of “free” information that has already been calculated at previous time steps.

A higher-order Adams-Basforth method is said to have a small *truncation error*, as it includes many terms in the Taylor's series expansion. This means that the error introduced at each step by the use of a finite value of Δt will be small. However, the repeated use of these equations may produce a large *cumulative error* after many integration steps. This is because they depend on extrapolation from past values of f that may have little relevance for the future. These methods tend to give the best results in cases where $f(t)$ varies smoothly, without sharp changes. If $f(t)$ varies rapidly with time, a lower-order method may have a lower cumulative error.

Exponential Euler Method

The *exponential Euler* method (MacGregor 1987) is the default integration method for GENESIS simulations. The efficiency and accuracy of this method depends on the fact that typically encountered equations such as Eqs. 2.1 and 4.4 assume the form

$$\frac{dy}{dt} = A - By. \quad (20.6)$$

Although A and B may depend on y and t in very complicated ways, we will see that the special case where $f(t) = A - By$ results in a considerable simplification of the problem to be solved. For a time step Δt , we can approximate the solution at a time $t + \Delta t$ by

$$y(t + \Delta t) = y(t)D + (A/B)(1 - D), \quad (20.7)$$

where we define

$$D = e^{-B\Delta t}. \quad (20.8)$$

This result follows from the fact that there is an exact solution for the differential equation when A and B are constant. In this case, it may be verified by substitution that we can express the value of y at a time t_2 in terms of its value at an earlier time t_1 using the relationship

$$y(t_2) = y(t_1)e^{-B(t_2-t_1)} + \frac{A}{B}(1 - e^{-B(t_2-t_1)}). \quad (20.9)$$

In fact, A and B are usually *not* constants. However, if we assume that they change very little between the time $t_1 = t$ and $t_2 = t + \Delta t$, we may use this result to obtain the approximate solution given above. Although it is difficult to rigorously analyze the error introduced by this approximation, simulation results show that it is highly accurate for most models that contain active channels and only a few compartments. In these cases, it allows much larger integration steps than the methods discussed so far.

20.3.3 Implicit Methods

The methods used in Eqs. 20.3, 20.5 and 20.7 are called *explicit* methods because the new values are given explicitly in terms of functions of the old values.

Backward Euler Method

The *backward Euler* method is an example of an *implicit* method. In this case we use

$$y(t + \Delta t) = y(t) + f(t + \Delta t)\Delta t. \quad (20.10)$$

For implicit methods, the right-hand side of the equation involves a function of the new value of y , which has yet to be determined. Thus, Eq. 20.10 gives an implicit definition of $y(t + \Delta t)$, rather than an explicit expression that can be evaluated. This means that we need some additional method to solve the equations that arise at each step. It would seem that there would be no point in using such an implicit method. From a Taylor's series expansion, you should be able to verify that the truncation error in Eq. 20.10 is the same as that of Eq. 20.3, but is of opposite sign. However, we will see that the use of implicit methods can lead to a much lower cumulative error under certain conditions.

Crank-Nicholson Method

In addition to the backward Euler method, GENESIS allows the use of another implicit method, the *Crank-Nicholson* method. This method is based upon the trapezoidal rule of numerical integration. It uses an average of the forward and backward Euler methods in order to achieve a partial cancellation of errors. This occurs because the neglected second derivative terms are equal and opposite in the two approximations. The approximation is then

$$y(t + \Delta t) = y(t) + (f(t) + f(t + \Delta t))\Delta t / 2. \quad (20.11)$$

20.3.4 Instability and Stiffness

Some coupled differential equations present particular problems when they are solved by stepwise numerical integration. For example, the exact solution to the differential equation may have a well-behaved slowly varying solution $y(t) = g(t)$, which we would expect to be able to obtain with a reasonably large step size. However, the difference equation that is used to approximate the differential equation may have another solution of the form $y(t) = g(t) + C \exp(Bt)$, where B has a large magnitude. The initial conditions are such that $C = 0$, so we would expect the difference equation to give the same result. However, as a result of the finite step size, truncation error will cause small errors in the computed solution that have the effect of modifying the initial conditions so that C is not exactly zero. The only way to ensure that this rapidly varying spurious solution remains negligibly small is to use a very small time step in order to minimize the error in its computation. Thus, we are forced to use a time step that is appropriate for a time scale much shorter than that of the actual solution. Examples showing how this type of instability may arise are given by Acton (1970) and by Press, Flannery, Teukolsky and Vetterling (1986).

This sort of behavior leads to the definition of a *stiff* differential equation as one whose solution contains a wide range of characteristic time scales. It turns out that the coupled sets of equations used in compartmental modeling often possess this type of stiffness. Unless a very small time step is used, this can lead to numerical instabilities, resulting in wild

oscillations in the computed solution. For example, the equations describing compartments become stiff when you use compartments with a small size (e.g., in dendritic spines) and/or inject large currents. In such cases, if your time step is not small enough, the simulation will blow up if the current increases beyond a certain threshold (as, for example, during an action potential). In general, explicit methods are much more prone to this type of instability than implicit methods.

As we have seen in Chapter 5, a uniform passive cable composed of identical compartments of length l and diameter d is described by the more specific form of Eq. 2.1,

$$C_M \frac{dV_i}{dt} = \frac{d}{4R_A} \frac{V_{i+1} - 2V_i + V_{i-1}}{l^2} - \frac{V_i}{R_M}. \quad (20.12)$$

(Here, we have taken Eq. 5.28 in the absence of channel currents, and have used Eqs. 5.4–5.6 to express it in terms of the specific membrane capacitance and resistance and the specific axial resistance, C_M , R_M and R_A , respectively.) Mascagni (1989) has analyzed the solution to this equation using the forward Euler, backward Euler, and Crank-Nicholson methods. When the forward Euler method is used, the solution is numerically unstable for $\Delta t > 2R_A C_M l^2 / d$. Thus, instability can be a problem for compartments that have a small length l , unless the step size Δt is very small. However, the backward Euler method is stable for any value of Δt . Of course, numerical accuracy will suffer if Δt is too large, but the solution will not display the catastrophic instability produced by the forward Euler method.

As the Crank-Nicholson method involves an average of the forward and backward methods, one might expect it to be less stable than the backward Euler method. Although the analysis shows that it is also stable for any value of Δt , simulation results show that it comes closer to the point of instability. Numerical solutions for stiff equations obtained with the Crank-Nicholson method often show spurious damped oscillations or over/undershoot when large values of Δt are used. Nevertheless, its greater accuracy per step (smaller truncation error) coupled with its moderately good stability make it a good choice for high precision calculations if the time step is sufficiently small.

Although it is an explicit method, the default exponential Euler method does not suffer the dramatic onset of instability shown by the forward Euler method (see Exercise 2). Interestingly, as you may verify in Exercise 3, it is nevertheless not as accurate as the forward Euler method with the same step size when there are many small compartments. Although it gives fairly accurate results for integration of the channel conductance equations, and is a good choice for simple cell models with few compartments, it will require a much smaller time step than the backward Euler or Crank-Nicholson methods if the equation to be solved is stiff. The lesson to be learned from this analysis is that you should choose a numerical method that is appropriate for the simulation to be performed, and should *always* experiment with different values of the integration step size before trusting your results.

20.3.5 Implementation of the Implicit Methods

So far we have neglected the question of how one implements an implicit method. The right-hand sides of Eqs. 20.10 and 20.11 involve functions that depend on the unknown quantity on the left-hand side. In general, we would have to use iterative methods, such as *predictor-corrector* or Newton's methods (Acton 1970) in order to find the solution.

Equation 20.12 and its more general form Eq. 2.1 each have the convenient property that the right-hand side of the equation involves the unknown membrane potential in the compartment and that in the immediately adjacent compartments. Thus, the matrix that represents the coupled sets of equations to be solved is a tridiagonal matrix — a sparse matrix with non-zero elements along the diagonal and just above and below the diagonal. This greatly simplifies the problem of solving the matrix equation. A method due to Hines (1984) provides a very efficient way to solve these equations when the coupled equations describe a branching tree-like structure (such as the dendrites of a neuron) without closed loops. In GENESIS, the Hines method is embodied in a special object **hsolve** which is used in conjunction with the backward Euler and Crank-Nicholson methods.

20.4 The *setmethod* Command

The *setmethod* command is used to specify the numerical integration method to be used in a GENESIS simulation. It takes an optional argument to specify the path to the elements to which the method will apply, followed by an integer identifying the integration method to use. If the path is omitted, the specified method will be applied to all currently existing elements in the simulation. (Any subsequently created elements will use the default method, however.) For example,

```
setmethod 2
```

will cause method 2 (the second order Adams-Bashforth method) to be used for all elements in the simulation. If this command is followed by

```
setmethod /cell 0
```

method 0 (exponential Euler) will be used for all elements in the */cell* element tree, and the rest of the simulation elements will be treated as before. Just as the *useclock* and *setclock* commands let us pick an integration step that is appropriate to particular simulation elements, we can use *setmethod* in this way to choose the most appropriate integration method. Table 20.1 gives the integers that correspond to the currently implemented GENESIS integration methods. The two implicit methods, methods 10 and 11, must be used in conjunction with an **hsolve** object, as described in the next section.

<i>Method number</i>	<i>Description</i>
-1	Forward Euler
0	Exponential Euler (default)
2	Adams-Bashforth 2nd-order
3	Adams-Bashforth 3rd-order
4	Adams-Bashforth 4th-order
5	Adams-Bashforth 5th-order
10	Backward Euler
11	Crank-Nicholson

Table 20.1 Presently available GENESIS numerical integration methods.

20.5 Using the **hsolve** Object

The **hsolve** method and its associated GENESIS object implement the Hines method of solving the equations for branched neuronal structures. By using **hsolve** with the backward Euler or Crank-Nicholson integration method, you will be able to use much larger integration steps in simulations of multi-compartmental models. In addition, the **hsolve** object provides a highly optimized code besides the use of the Hines implicit integration schemes. Even with the same size integration step, the **hsolve** method results in a speedup of at least 50% over the usual exponential Euler method.

However, this greater speed comes at a price. The matrix methods that are used to solve the coupled differential equations are incompatible with the object-oriented nature of GENESIS, in which elements communicate only by the exchange of messages. In order to carry out the solution, we create an **hsolve** element that takes over the calculations which are performed by a cell or other tree of linked compartments. The entire cell then acts as a single object, since the individual compartments within the cell are no longer responsible for their own computations. However, this is done in such a way as to preserve the *illusion* of object orientedness. Within certain limits, you may continue to set element fields and create and delete messages between objects. The action of the **hsolve** object on the simulation elements has been compared to one of those 1950's science fiction films in which alien beings have taken over your friends. Outwardly, they seem normal. Only occasional departures from normal behavior reveal that something very different is in control (Bhalla 1990).

In order to make the **hsolve** method as efficient as possible, only a few of the GENESIS objects can be taken over by **hsolve**. These are the ones that typically account for the greatest computational load in a simulation, and are most likely to lead to numerically stiff equations. At present, the objects that are supported by **hsolve** are the **compartment**, **tabchannel**, **tab2Dchannel**, **tabcurrent**, **synchan**, **spikegen**, **Ca_concen**, **nernst**, **ghk**, **difshell**, **fixbuffer** and **difbuffer** objects. Other types of elements continue to be treated as they were before the **hsolve** element took over the cell.

The Hines algorithm is only applicable to branched structures that do not form closed loops of interactions between their elements. This prevents it from being applied to gap junctions (Sec. 19.6). This also means that the method should be applied to individual cells in a network, but not to the network as a whole, unless the network has no loops of interactions.

Because of these various restrictions and the loss of flexibility in setting fields and changing messages, we suggest that you first build and test your model using the default exponential Euler method. Once it is working properly and is unlikely to be changed, you may add the statements necessary to implement the **hsolve** method.

Future releases of GENESIS will provide a *readsolve* command. This command reads the same files as the *readcell* command and creates the corresponding **hsolve** element, without creating the other elements in the cell tree. As a consequence it uses less memory than the standard way of invoking **hsolve**, but the special *findsolvefield* function must be used to output simulation values.

20.5.1 Modes of Operation

The **hsolve** object has various modes of operation, referred to as *comptmodes*, *chanmodes*, *calcmodes* and *storemodes*. The *comptmode* field is a flag that determines the way compartment computations will be performed. For best performance, this field should be set to 1 (the default). This, however, increases memory use significantly. If you run out of memory, try setting *comptmode* to zero before setting up the **hsolve** element.

The *chanmode* field controls channel computations. At present, there are four *chanmodes* available in GENESIS, labeled with the integers 0 through 4. After an element is created from the **hsolve** object, the mode to be used is selected by setting the *chanmode* field in the element. In general, the higher mode numbers are faster, but place more restrictions on your ability to modify the simulation. All of the modes have the same accuracy, and thus allow you to use comparable time steps. In general, these will be an order of magnitude larger than the maximum step size allowable for the default exponential Euler method.

chanmode 0

This is the default mode of operation. Although it uses the least amount of memory, it is also the slowest. To implement the Hines method, it takes over the actions of compartments only, computing all other object types as before. As a consequence, all computed fields of the original elements are updated, and all user-setable fields may be set, just as before the element was taken over by the **hsolve** element. This means that you can add and delete outgoing messages to compartments or other elements whenever you like and easily change parameters during the course of the simulation.

One significant limitation is that you cannot add or delete AXIAL, RAXIAL or CHANNEL messages once the **hsolve** element has been created. If you can live with these limitations, *chanmode* 0 is the easiest one to use. It is also the most compatible mode of operation, and is guaranteed to work with any future new object type.

chanmode 1

This *chanmode* uses optimizations for the computation of **tabchannel** current to the **hsolve** computations. It has the same limitations as *chanmode* 0, but achieves a large increase in computation at the cost of increased memory use. This is the preferred *chanmode* if you employ **tabchannels** in your model and do not want to use *chanmodes* 2 through 4.

chanmode 2

The *chanmodes* 2 through 4 achieve a significant speedup at the expense of greatly increased memory usage. Under these modes, the original elements describing channels, concentrations, etc. are not used at all. Instead, all simulation parameters are stored in a huge array, called the *chip-array*. This reorganization optimizes the speed of the use of CPU cache memory. If your computer has limited memory and is forced to perform increased swapping to disk after changing to *chanmode* 2, this may negate the speed advantage. These *chanmodes* also assume that *comptmode* is set to 1 and will change the *comptmode* field if necessary. As a consequence of the use of the chip-array, the element tree of your cell (or other element tree to be taken over by hsolve) must not contain any non-hsolvable elements when these *chanmodes* are used. However, within the limitations described below, messages may be exchanged with non-hsolvable elements that lie outside the hsolved element tree.

chanmodes 2 through 4 require integer exponents for **tabchannel** and **tab2Dchannel** gate variables. In addition, the maximum exponent allowed is 6. Incoming and outgoing messages from the disabled elements are supported, providing that they are established *before* setting up the **hsolve** element. If you try to add new messages afterwards, they will go to the old disabled elements only, and will be ignored. Likewise, using *deletemsg* to remove an existing message after the set up will have no effect.

Under these modes, you can no longer assume that all the fields of the elements that are taken over by hsolve will be updated. This will make it harder to display these fields graphically, or to output their values to a file. However, the *Vm* field of all compartments will be automatically updated when using *chanmode* 2. Any new outgoing messages to non-hsolved elements are from the original objects. Thus, whether the message works depends on whether their fields get updated. You can always establish a new outgoing message, but it may not give current information. You can prevent this problem from occurring by using the **hsolve** element itself as the source of outgoing messages, using the *findsolvefield* function.

Another consequence of this “takeover” is that *setfield* commands will be ignored until you perform a *reset*. The GENESIS Reference Manual describes how this restriction may be overcome with the **hsolve** object’s HPUT and HRESTORE actions or by using the *findsolvefield* function.

chanmode 3

This mode is very much like *chanmode 2*, except that elements are automatically updated if they had outgoing messages to non-hsolved objects prior to the setup. This means that you can plot or otherwise access fields other than *Vm*, but the *Vm* is not automatically updated as in *chanmode 2*. On the other hand, if any new messages are added after setup, the old values are sent. Note also that several fields (e.g., *Gk*, *Ik*, *Ek* and *Im*) are not available for output in *chanmode 3*, so you have to use *chanmode 4* if you want to output these fields

Mode 3 offers a little more efficiency because it allows you to specify the clock that is used to update fields in the original elements. This means that if you want to send the value of a field to a graph with a PLOT message, the element fields can be updated at a slower rate than the one used by the simulation clock. Of course, the elements that have been taken over by **hsolve** will exchange messages at the faster rate specified by the simulation clock. The update clock is specified by setting the *outclock* field of the **hsolve** element. For example,

```
setclock 1 10*{dt}
setfield /cell/solve outclock 1
useclock /graphs/conductance_plot 1
```

Note that if you want to output *Vm* only, but from multiple compartments, or to create and delete plots of *Vm*, you should use *chanmode 2*.

chanmode 4

This mode is identical to *chanmode 3*, but allows you to output additional fields at the cost of a reduction in computation speed. In *chanmodes 2* and *3*, the **hsolve** element “forgets” values like *Ek* and does not compute values like *Ik* and *Im* because they are not needed during the present or next integration step. As a consequence, the *Ek* and *Ik* fields of a disabled channel element cannot be updated. If you had specified an outgoing SAVE or PLOT message for one of these fields prior to setup, the output would always be zero. In *chanmode 4* the values of *Ek*, *Ik* and *Im* are computed and stored in a *givals-array* so that the proper values will be available for SAVE or PLOT messages, when used with *findsolvefield*. Additionally, in *chanmode 4* a field *leak* is available for each compartment, which gives the leak current flowing through the compartment *Rm*.

To give an idea of the effect of *comptmodes*, *chanmodes* and *calcmodes* on computation speed, we provide a comparison of computation times in seconds for 1000 steps of the standard 1600 compartment Purkinje cell model of De Schutter and Bower (1994a) on a HyperSparc platform in Table 20.2.

		<i>calcmode 0</i>	<i>calcmode 1</i>
comptmode 0	chanmode 0	n.a.	137.9
comptmode 0	chanmode 1	n.a.	82.8
comptmode 1	chanmode 0	n.a.	139.1
comptmode 1	chanmode 1	n.a.	76.4
comptmode 1	chanmode 2	23.0	24.0
comptmode 1	chanmode 3	21.4	21.8
comptmode 1	chanmode 4	26.9	27.3

Table 20.2 Comparison of computation times (in seconds) for various combinations of *chanmodes*, *comptmodes* and *calcmodes*. 1000 steps of simulation were performed for a 1600 compartment Purkinje cell.

storemode

With chanmode 4, the *storemode* field allows the output of total currents and conductances from a compartmental model. This technique was used extensively by Jaeger, De Schutter and Bower (1997) to study the role of voltage-gated currents in the control of Purkinje cell spiking. For each type of voltage-gated channel in the model, the total currents or conductances are the sum of the corresponding *Ik* or *Gk* fields for all compartments where the channel is present. This assumes that these channels have the same name in each compartment. Total currents are computed if *storemode* is set to 1, and total conductances if it is set to 2. These are stored in an array called *itotal*. When the **hsolve** element is set up, a message will be output giving a list of channel names and corresponding *itotal* indices if the *silent* command has previously been given with a negative argument. Alternatively, you may start GENESIS by giving the *genesis* command with the *-silent* option and a negative value.

calcmode

The *calcmode* field affects operations for *chanmodes* 2 through 4 only. It is initialized to 1 (LIN_INTERP), which is the recommended mode. In the LIN_INTERP mode, all values obtained from lookup tables will be interpolated, ensuring a higher accuracy. The *calcmode* exists mainly to provide backward compatibility, as older versions of the **hsolve** object in GENESIS versions 2.0 and earlier did not use interpolation, which is equivalent to setting *calcmode* to 0 (NO_INTERP).

20.5.2 Rules for Table Dimensions

The *chanmodes* 2 through 4 of the **hsolve** object also reorganize the lookup tables present in **tabchannel**, **tab2Dchannel** and **tabcurrent** objects to achieve an extra increase in computation speed. However, this works only if *all* lookup tables are of the same dimensions. Specifically, the *xdivs* field in all tables should be identical and if **tab2Dchannels** are used, the *ydivs* should equal the *xdivs*. Moreover, all voltage-indexed tables should have identical *xmin* and *xmax* values. The same is true for all concentration-indexed tables. The **hsolve** element will alert you during the setup phase if these rules are not obeyed.

In practice, when **tab2Dchannels** are used in the simulation, all tables will have a few hundred *xdivs* instead of a few thousand because otherwise you will rapidly run out of memory. Because of the small size of the tables it is very important to use the **LIN.INTERP calcmode**. Obviously, you should not use the **TABFILL** action on such small tables, but instead compute every entry of the table.

20.6 Setting up hsolve

The preferred location of the **hsolve** element is as the root element of the cell. Although the actual location of the **hsolve** element within the tree of elements does not matter in GENESIS 2.1, it will do so in future GENESIS versions. If you want to use *chanmodes* 2 through 4, you should also organize the tree of elements describing your model properly:

1. The **hsolve** element should be the root of the cell element tree.
2. The next level of the tree should contain compartments only.
3. All channel, concentration, etc. objects should be children or grandchildren of the compartment to which they are attached.
4. To make the *findsolvefield* command behave properly, all grandchildren of a particular compartment should have different names.
5. The cell element tree should not contain any elements that are not computed by the **hsolve** object except for *neutral* elements.

The **hsolve** method may be applied by following these steps:

1. Create the **hsolve** element and name it as you would name the cell using a **neutral** object, e.g., */cell*. If you plan to use the **readcell** command to create */cell* you should use the *-hsolve* option to achieve the same effect:

```
readcell {filename} /cell -hsolve
```

2. If you do not use `readcell`, create the cell or other structure that is to be treated with the **hsolve** element using it as the root element. For example,

```
create hsolve /cell
create compartment /cell/soma
create tabchannel /cell/soma/Na_channel
...
...
```

would be used to have an **hsolve** element called `/cell` compute the `/cell` element tree.

3. If your **hsolve** element is not the root element of the cell, you should set the path to the elements that will be treated with **hsolve**. This may be best done using a wildcard specification such as

```
setfield /cell path /cell/##[] [TYPE=compartment]
```

to set the *path* field of *solve* to an expression describing all **compartment** elements in the `/cell` hierarchy. Any subelements of the compartments that are treatable by **hsolve** will be taken over by **hsolve** as well. Thus, it is not necessary to list any **tabchannel** or **Ca_concen** elements that are associated with the compartments.

4. If you use a *chanmode* other than the default *chanmode* 0, set the *chanmode* field of the **hsolve** element to the mode number. For example,

```
setfield /cell/solve chanmode 3
```

5. Then, tell the **hsolve** element to create all the solution arrays and tables by calling its **SETUP** action:

```
call /cell SETUP
```

6. Set the integration method to be used with the *setmethod* command. For example,

```
setmethod 11
```

will cause the Crank-Nicholson method to be used for all elements that are taken over by **hsolve**. The default is the backward Euler method. Those elements that are not treatable by **hsolve** will continue to be treated by the method which was previously in use, usually the default exponential Euler method.

7. Finally, it is essential to call *reset* after setting up an hsolver, so that the process list gets updated.

```
reset
```

Now you are ready to try running the simulation, starting with the original step size and decreasing it until errors begin to creep in. Often, the **hsolve** method will allow you to use time steps that are an order of magnitude larger than those required for the exponential Euler method.

Sometimes you may need to modify certain element fields or messages during the course of the simulation. If the *chanmode* you are using does not allow this, the easiest solution is to delete the **hsolve** element, make the changes, and then recreate the element by repeating the steps given above.

20.6.1 The **findsolvefield** Function

The **findsolvefield** function is used together with *chanmodes* 2 through 4 to access the **hsolve** arrays directly to output field values instead of having to use the old disabled elements. For example, to output the *Gk* field of *Na_channel* from the **hsolve** element */cell*:

```
call /cell SETUP
...
create disk_out /output
addmsg /cell /output OUTPUT {findsolvefield /cell /cell/soma/Na_channel Gk}
...
reset
```

The **findsolvefield** function returns a string containing the **hsolve** field corresponding to the disabled element field. The syntax of **findsolvefield** is:

```
findsolvefield hsolve element_path field
```

The **findsolvefield** function is always safe to use for output. Of course, whether you can output a particular field will depend on the *chanmode* used. The **findsolvefield** function will alert you if the field is not available. In general, it is not advisable to use the **findsolvefield** function for *setfield* commands, except in this particular example:

```
// inject 5 nA of current
setfield /cell {findsolvefield /cell /cell/soma inject} 0.5e-9
```

It is generally dangerous to use **findsolvefield** in a *setfield* command because the **hsolve** object modifies many field values before storing them in its arrays.

20.6.2 The DUPLICATE Action

If you are creating a network of cells, each cell must have its own **hsolve** element. If the cells are identical, you can save a great deal of memory by making copies of the original cell (and attached **hsolve** element). Then, use the DUPLICATE action for each of the copied **hsolve** elements. This conserves memory by allowing many of the tables in the original **hsolve** element to be used for the copies. For example, if */cell2* is to be created from */cell*, use

```
// this copies the element tree including the hsolve root element
copy /cell /cell2
call /cell DUPLICATE /cell2
```

As we described in Chapter 18, large networks may be created with a *for* loop, or with one of the specialized functions such as *createmap*.

20.7 Experiments with the **hsolve** Object

This experiment and those in the following exercises reveal some of the advantages and restrictions of the various *chanmodes*. There is a bewildering variety of combinations of *chanmodes*, other **hsolve** field settings, and actions that you may call. Performing these experiments will not only help you to be sure that you understand their documentation, but it will also reveal any changes that may have occurred in later versions of GENESIS. The **hsolve** object is under continuous development, so you should consult the **hsolve** documentation that accompanies your GENESIS distribution if you intend to use any of the advanced features of **hsolve**. It is possible that some of the restrictions described above will have been removed.

Begin by creating a 20-compartment passive cable. Provide current injection to the first compartment by setting its *inject* field and generate plots of the membrane potential in the first and last compartments. The script for *tutorial4.g* from Chapter 15 would be a good starting point, as it has the graphics and control widgets you will need, as well as a function *makecompartment* for creating the compartments. The parameters used in this simulation for the single dendrite compartment will be suitable. The easiest way to create the cable of linked compartments is to borrow a function from the *Cable* tutorial, which we used in Chapter 5. The script *addcable.g* defines a function *make_cable* that uses a *for* loop to construct a cable with a specified number of compartments. Use this function to construct the cable in place of the two compartment neuron used in *tutorial4.g*. You will also find it useful to provide a dialog box for changing the integration step size.

Test your simulation using the default exponential Euler method, without creating an **hsolve** element. If you have any doubts that it is behaving properly, compare it with the results of the *Cable* simulation, using the same parameters. Then set up an **hsolve** element

and run the simulation using either the backward Euler or Crank-Nicholson method and *chanmode* 0. How much larger a time step can you use for the same accuracy?

After verifying that the simulation also works under the other *chanmodes*, see what happens if you interactively add new PLOT messages under the various modes. You may use the INJECT message to provide injection pulses to a compartment by using the function *inject_compt* from the *inputs.g* script used in the *Cable* simulation. If you prefer, you may provide a constant injection by creating a **neutral** element, setting its *x* field to the desired injection current, and sending it to the compartment with an INJECT message. Do the various *chanmodes* behave as described in Sec. 20.5.1?

20.8 Exercises

1. Add a soma with voltage dependent Na and K channels to the cable that you created in the experiment described above. As you will have to use **tabchannels** instead of **hh_channels** with **hsolve**, create your channels from functions defined in the *neurokit/prototypes* script *mitchan.g*, rather than *hhchan.g*. Describe the results under the exponential Euler and the Crank-Nicholson methods for various step sizes, and present some evidence that your simulation is working as expected.
2. Run the *Cable* simulation from Chapter 5, using a cable of 20 identical compartments, each having a diameter of $2 \mu\text{m}$ and a length of $100 \mu\text{m}$. Select the forward Euler method from the numerical methods menu and vary the step size *dt* in order to determine the point at which numerical instabilities begin. Take some care to locate this point fairly precisely. How does this value compare with the value of Δt predicted in Sec. 20.3.4? (If you prefer, you may use the simulation that you created in this tutorial instead. Rather than setting up the **hsolve** element, use *setmethod* to set the method to “-1”.)
3. Use the backward Euler method with the cable described in the previous exercise. Experiment with increasing values of Δt , until you start noticing slight changes in the plotted results. Determine the largest value of Δt that should be used for accurate results. Then try the forward Euler method, exponential Euler, and the second- through fourth-order Adams-Bashforth methods. For each one, list the largest value of Δt that produces a plot indistinguishable from that produced by the backward Euler method. What do you conclude about the relative accuracies of these methods when applied to this problem?

Chapter 21

Large-Scale Simulation Using Parallel GENESIS

NIGEL H. GODDARD AND GREG HOOD

21.1 Introduction

PGENESIS is a parallel form of GENESIS that enables simulation of very large models. Simulation models are critical for integration of behavioral data with anatomical and physiological data. Although explanations of behavioral data are possible without resort to neural simulation models (Chomsky 1957, e.g.), those integrative accounts that make contact with the anatomical and physiological data require large-scale simulation models at the neural level. The scale of the models required can be seen in theories about the function of the hippocampus in learning and memory (McClelland and Goddard 1996, Levy 1996). These theories assert that statistical properties of firing rates, synaptic transmission efficiencies, and connection structures are crucial in explaining information processing in the hippocampus. The validity of these statistical properties is conditioned on sufficient sample sizes that cannot hold if the model scales down the real system by more than one or two orders of magnitude. Even scaling down by two orders of magnitude leaves us with very large models that, as we shall see, go beyond the capabilities of existing simulation environments.

Two developing data acquisition methodologies are driving this interest in larger scale models. Technologies for imaging the nervous system, particularly functional magnetic resonance imaging (Belliveau, McKinstry, Buchbinder, Weisskoff, Cohen, Vevea, Brady and Rosen 1991), allow us for the first time to test hypotheses embodied in systems level

models and to relate these models to behavior. Multi-electrode recording devices (Wilson, McNaughton and Stengel 1992), which are now migrating from the small number of originating labs to a larger group of users in diverse labs, are already delivering individual spike data for over one hundred cells simultaneously, allowing modelers to refine hypotheses about information representations employed in particular systems.

Effective use of a parallel computer for simulation requires that the problem be partitioned in such a way as to minimize the overhead associated with communication and synchronization between processors, while at the same time balancing the amount of work done by each processor. Overhead is minimized by reducing the communication between components of the simulation that reside on different processors, and by maximizing the extent to which the components of the simulation on different processors can proceed without synchronizing. There are two characteristics of communication hardware (e.g., Ethernet) that are referred to in this chapter. *Bandwidth* is the effective rate at which data can be sent over the medium. *Latency* is the time between the transmission of data by one processor and their reception by another. Two classes of neural simulations can often exhibit low overhead:

1. **Parameter search.** Global optimization algorithms for fitting a parameterized model to data require that many parameterizations be evaluated. In neural models evaluation of a particular parameterization is often best achieved by running the model. There are many optimization algorithms that can be parallelized so that many parameterizations are evaluated simultaneously, including parallel genetic algorithms (Collins and Jefferson 1991) and parallel simulated annealing (Azencott 1992). Each parameterization of the model can be run on a separate processor. The communication costs are small: the parameters must be transferred on startup, and the fitness value returned at the end of the evaluation run of the model. Synchronization costs can be very low because models are run independently. The amount of synchronization required depends on the particular search strategy, as discussed further in Sec. 21.7.
2. **Network models.** Network models often exhibit two characteristics that closely match the underlying hardware of parallel platforms, if we assume a partition of the model that keeps the compartments of each neuron on a single processor so that the communication is exclusively in the form of spikes. First, spike rates are low compared with the time step for integration within the cell, thus communication bandwidth is low as few spikes need to be communicated on each time step. Second, axonal delays are typically one or two orders of magnitude greater than the time step, so that cells need not be simulated in lock step. Simulation time on different processors can differ as long as every spike is delivered to its destination within the axonal delay period. These characteristics match those in parallel platforms, in which bandwidth is often limited and latency is often high.

In contrast, distributing a simulation of an electrotonically connected model over many processors will incur much larger overhead. This is because at each time step the processors must exchange data values and, implicitly, synchronize. There is extensive communication, and processors must wait for all to finish a time step before they can proceed to the next step. In this chapter will introduce the use of PGENESIS for the two classes of models enumerated above: parameter search and network models. We also discuss hybrid simulations in which parameter search is performed on a network model that itself runs in parallel.

21.2 Classes of Parallel Platforms

The three major classes of parallel platforms are (1) networks of workstations (NOWs), (2) symmetric multiprocessors (SMPs) and (3) non-uniform shared memory massively parallel processors (NUMA MPPs). In addition there are hybrid versions of NOW and MPP in which each node of the machine is itself an SMP. The two most important efficiency questions when deciding which platform to concentrate on are: how well its communication characteristics match those needed by the simulation task; and how the architecture scales up to the number of nodes that the model can use.

As stated before, the important communication characteristics are bandwidth and latency, and of these latency is usually more critical for neural models. Low latency and high bandwidth is the goal. To simplify vastly, NOWs typically have relatively high latency and low bandwidth. Thus they are suitable for coarse-grained parallel tasks such as a parameter search task where evaluation of a single individual takes an appreciable amount of time. They may also be suitable for network models with very long lookahead that may not be adversely affected by high latency. Other factors to consider with a NOW platform are whether one has exclusive access to the processors and how much disk traffic the simulation generates. Without exclusive access, it is very hard to partition a network model efficiently. NOW platforms are typically not set up well for massive transfers to and from shared disk for many processors.

SMP platforms (e.g., Origin, Convex, Ultrasparc) have very low latency and high bandwidth for small to medium numbers of processors (e.g., up to 16), but these characteristics do not scale well to a high degree of parallelism because the underlying hardware communication medium — a shared bus — does not scale. Nevertheless they can be very effective for development of small to medium scale PGENESIS models and are often easier to use than MPPs or NOWs.

NUMA MPP machines (e.g., Cray T3E) are the ideal platform for the largest, most highly parallel PGENESIS tasks. The latency and bandwidth characteristics are vastly better than NOW and approach SMP, and the architecture can scale into the hundreds of processors on parameter search tasks, and into the tens of processors for well-distributed network models. These machines are at the high end of high performance computing and

so are designed to balance processor speed with memory latency and bandwidth and disk latency and bandwidth. The largest network models will almost certainly require them. However, the NOW or SMP platforms may be more cost-effective for large-scale parameter search tasks.

21.3 Parallel Script Development

Script development for PGENESIS is an exercise in parallel programming where many processes are running simultaneously. Each process (called a *node*) is running one instance of a single parallel script. For modelers familiar with serial programming, including all GENESIS users, the transition to parallel programming requires some conceptual leaps. For example, one can no longer assume that because statement B in a script comes after statement A that B will be executed after A. The order depends on which nodes the statements are executed by, and what synchronization events (e.g., barriers) occur between them.

We have found that PGENESIS simulations are best developed in the following order.

1. For all models, develop and debug the single cell prototypes using serial GENESIS.
2.
 - (a) For network models, decide how the network should be partitioned, i.e., which GENESIS elements should go on which nodes. It is best to implement the scripts in a scalable fashion so that the number of nodes can be varied by changing a run-time parameter.
 - (b) For parameter search tasks, develop the scripts to run and evaluate a single individual, and the scripts that control the optimization. The optimization scripts should be parameterized so that they will run with any number of nodes.
3. First try out your scripts on a single processor — a desktop workstation is often most convenient for this stage. Make sure that your scripts run correctly using the minimum number of nodes (a single node, if possible). Also, be sure the scripts will run in the background, without XODUS or any interactive input.
4. Continue to use the single processor platform, but increase the number of nodes in the simulation. This will show up errors related to the partitioning of the problem across more than one node.
5. Run the scripts on a multiprocessor platform with a small number of processors. For example, a small symmetric multiprocessor or a small number of networked workstations. This will show up errors due to assumptions about execution order.
6. Run the full scale simulations on the largest machine you need and have access to. By this time your scripts should be well-debugged. Typically debugging is difficult on

the largest platforms, so it is prudent to do as much debugging as possible on smaller machines.

21.4 Script Language Programming Model

A PGENESIS simulation consists of a set of independent processes (*nodes*) that can communicate via script language commands and can cooperate in a simulation via GENESIS messages between elements residing on different nodes. Serial GENESIS forms the core of each of these processes. As in serial GENESIS, execution of a simulation including setup and stepping, is controlled by scripts. Use of XODUS is discussed in Sec. 21.8. However, script programming for PGENESIS introduces additional complexities. It is critical that a user of PGENESIS review and understand these issues before attempting to run PGENESIS.

21.4.1 Parallel Virtual Machine

PGENESIS is built on top of the Parallel Virtual Machine (PVM) software system (Geist, Beguelin, Dongarra, Jiang, Manchek and Sunderam 1994), which provides the illusion of a parallel platform. The PVM system may run on a single CPU or multiple CPUs, possibly of different types. In the rest of this chapter, when we refer to the “parallel platform” we mean the illusion provided by PVM. When we refer to the “parallel machine,” we mean the physical set of CPUs and the network connecting them on which PVM is running. An executing PVM program consists of user processes, typically one per CPU, which communicate via the PVM daemon that runs on each participating CPU. In PGENESIS, each user process is an independent GENESIS simulation, which we call a *node* of the parallel simulation. Nodes are uniquely identified by a node number (consecutive integers starting at zero). These nodes may be grouped into *zones* — nodes within a zone have their simulation time kept more or less synchronized, whereas simulations in different zones may run relatively independently. Thus, a parameter search algorithm would typically run many simulations independently with each node in a separate zone, whereas a large network model would typically run with all nodes in a single zone.

21.4.2 Namespace

PGENESIS currently provides a private-namespace programming model. This means that each node has no knowledge of the elements that reside on other nodes. This implies that every reference to an element on another node must specify the node explicitly. It is envisioned that a shared-namespace programming model will be implemented eventually. This will allow nodes within a zone to reference elements on other nodes in the zone without specifying the node number. To ease upgrade of parallel models to the shared-namespace

paradigm, it is recommended that element names within a zone be unique. If this recommendation is not adhered to, there will be naming conflicts if a model wishes to take advantage of the shared-namespace capability when it becomes available.

21.4.3 Execution (Threads and Synchronization)

The main thread (i.e., flow of control) on each node is that which reads commands from the script file (or keyboard, if the session is interactive). PGENESIS provides limited capabilities for this thread to create new threads on any node. On each node, the threads are pushed onto a stack with the main thread at the bottom of the stack. Only the topmost thread may execute, and when it completes it is popped off the stack so that the next thread down can continue. Threads ready to execute are *not* guaranteed to execute: if the topmost thread is blocked or looping, no ready thread lower on the stack can continue.

An executing thread is guaranteed to run to completion (assuming it does not contain an infinite loop or block on I/O) as long as it executes only local operations, i.e., no operations that explicitly or implicitly involve communication with other nodes. The command descriptions below include specification of local or non-local status. In addition, simulation steps and reset are by definition non-local operations if there is more than one node in the zone. Users are strongly encouraged to use only local operations in child threads whenever possible. Users need to be very careful about thread creation to ensure that *deadlock* (when no thread can continue) does not occur.

PGENESIS provides facilities for blocking and non-blocking thread creation, usually used to execute commands on nodes different from the one on which the script is being executed. (“remote” nodes). When a thread (including the main script) initiates a blocking remote thread (also known as a *remote function call*), it waits until the thread completes before continuing. When a thread initiates a non-blocking remote thread (an asynchronous thread), it continues immediately without waiting for termination of the thread. While a thread is waiting, the node can accept a request for thread creation arriving from any node (including itself). This new thread is pushed on the thread stack and executed, so that the original waiting thread does not continue until the new thread has completed.

Scripts running on different nodes can synchronize via several different synchronization primitives. The simplest, a *barrier*, causes every node to wait at the *barrier* statement until all other nodes have reached that point in the script. There are two types of barriers, one that involves all nodes in a zone, the other involving all nodes in the parallel platform. By default there is an implicit zone-wide barrier before a simulation step is executed, although this can be disabled.

When a script requests that a command be run asynchronously on another node, it initiates a child thread of control on the other node. The child thread runs asynchronously with its parent. The parent can request notification or the child’s result when the child completes, and can wait on that notification or result (a “future”), and this is the only way to ensure

asynchronous child threads have completed. Threads do not block for child completion before each simulation step, nor at a barrier. It is easy to reach deadlock if the creation and execution of threads are handled carelessly.

If a node initiates several child threads on a particular remote node, these are guaranteed to commence (but not necessarily complete) execution in the order in which they were initiated. A thread is guaranteed to execute eventually as long as no preceding thread (1) enters a loop that only executes local operations, or (2) blocks indefinitely because of deadlock. Once execution of a thread begins, it runs to completion without interruption as long as it only executes local operations.

21.4.4 Simulation and Scheduling

PGENESIS provides the ability to set up a GENESIS message between two elements on different nodes (a remote message), provided the nodes are in the same zone. Data are physically transferred from one node to the next at the beginning of a simulation step. This means that there is no transfer of data between elements on different nodes within a single time step, which has ramifications for the schedule. (The GENESIS Reference Manual contains a description of simulation schedules.) PGENESIS guarantees that execution on a parallel platform will be identical to that on a single processor if and only if there are no remote messages for which the source object precedes the destination object in the schedule. (We assume that every node in a zone has the same schedule.)

21.4.5 Node-Specific Script Processing

Each node executes a single main script common to all nodes. However, it is common to need node-specific script processing. A node is assigned a unique identification which is available to the script it is processing via the *mynode* and *myzone* commands. If execution of script statements is conditional on the node ID, then different nodes can execute different scripts. For example, if the main script (henceforth, *main.g*) executed by all nodes contains this script fragment.

```
early-statements...
if (mynode == 0)
    include node0.g
else
    include node1.g
end
later-statements...
```

then each node 0 will execute *early-statements*, followed by statements in script *node0.g*, followed by *later-statements*. All other nodes will execute *early-statements*, followed by statements in script *nodeX.g*, followed by *later-statements*.

21.4.6 Asynchronous Simulation

The PGENESIS nodes usually operate asynchronously, so instantaneous position in a script varies between nodes. In the preceding script fragment, one node still could be executing *early-statements* while another is already executing *later-statements*. This means, for example, that one node may have completed element creation and be adding messages while another node is still creating elements. If the first node tries to send a message to an element on the second node, it may find that the target element has not been created. Modelers can control this behavior through the judicious use of barriers. A *barrier* is a script statement that must be executed by all nodes before any node can continue past the *barrier* statement. For example, the following main script fragment will guard against the problem of attempting to send a message to an element not yet created:

```
create_elements(arg1, arg2, ...)
barrier
create_messages(arg3, arg4, ...)
```

In this main script fragment, each node creates the required elements, then waits at the *barrier* statement. Only when all nodes have reached the barrier, and therefore created their elements, can any node continue on to create messages.

Some script commands result in implicit synchronization events. For example, by default, the nodes synchronize before executing a simulation step. The *reset* command also causes the nodes to synchronize.

21.4.7 Zones and Node Identifiers

Nodes can be grouped in zones when the simulation is started. Each node is in exactly one zone (by default, every node is in its own zone). The zones form a fixed partition of the parallel platform. The motivation for using zones is to allow different parts of the simulation to run asynchronously (uncoordinated) even during simulation steps. For example, in a parameter search application, one might wish to run many instances of a four-node model in parallel. Each instance uses four nodes that must run synchronously, but the instances need not be coordinated (except at start and finish). Thus, we can run each instance in a separate zone, each zone containing four nodes. Zones are uniquely identified by consecutive integers starting at zero. The nodes within a zone are uniquely identified with a node number (consecutive integers starting at zero).

Node identifiers are of the form “`p.q`” where `p` is the number of the node within zone `q`. Node identifiers are used in commands that expect an element path which may be on a remote node (e.g., `raddmsg` in Sec. 21.4.10) and in remote function calls (see Sec. 21.4.8). The zone specification “`.q`” can be omitted, in which case the zone of the node executing the script is assumed. In network models there is often only a single zone, so that the zone specification can be omitted everywhere. In optimization tasks there is typically only one node in each zone, so that nodes are referred to with “`0.n`”.

A script accesses the node number and zone number of the node on which it is running with the `mynode` and `myzone` commands. For example:

```
echo I am node {mynode} in zone {myzone}
```

will print the node and zone numbers on which the script is running.

21.4.8 Remote Function Call

A script running on a node can execute a function or command on another node simply by appending “`@ID`” to the command, where `ID` is the identification string for the node. We refer to the node on which the script is running as the *issuing* node, and the remote node on which the function or command is executed as the *executing* node for the remote function call. For example, if node 3 in zone 1 executes the remote function call:

```
echo@0 hello from node {mynode} in zone {myzone}
```

then the issuing node is node 3 in zone 1, and the executing node is node 0 in the same zone as the issuer (i.e., zone 3). This command will cause “`echo hello from node 3 in zone 1`” to be executed on node 0 of zone 1. Notice that the evaluation of the commands `mynode` and `myzone` is performed on the issuing node, not the executing node. Argument evaluations are always performed on the issuing node.

Two special keywords can be used with the `@` operator: *all* means all nodes or all zones, depending on context; *others* means all other nodes or zones, depending on context. Here are some examples of how they can be used:

<code>step@all</code>	<code>// step every node in the zone</code>
<code>func@all.all</code>	<code>// call func on every node in every zone</code>
<code>func@3.others</code>	<code>// call func on node 3 in other zones</code>
<code>func@all.3</code>	<code>// call func on every node in zone 3</code>

In addition, node identifiers can be composed into a list separated by commas. For example

<code>func@1,3,5</code>	<code>// call func on nodes 1, 3 and 5</code>
-------------------------	---

This is used in the network model in Sec. 21.6.

The remote function call examples shown in this subsection are *synchronous*. The issuing node suspends execution of the script containing the remote function call statement until the executing node has completed the function call and returned the result. This is shown schematically in Fig. 21.1. In this figure we also show the execution of the script on node 1 as a solid line. Conceptually, there are just two threads of control active at any time. During the execution of the remote function call on node 1, both threads reside on node 1. In PGENESIS there is exactly one thread of control per node, as long as the *async* command is not executed, but by using remote function calls, some nodes may have no threads resident, and other nodes may have more than one resident thread.

PGENESIS users should be aware that the threads of control are not full, independent threads, able to suspend and resume arbitrarily. They are implemented in a stack-based system so that only the thread at the top of the stack can execute. When it completes it is popped off the stack and the preceding thread resumes. (It may immediately suspend again, but it is given the chance to resume.)

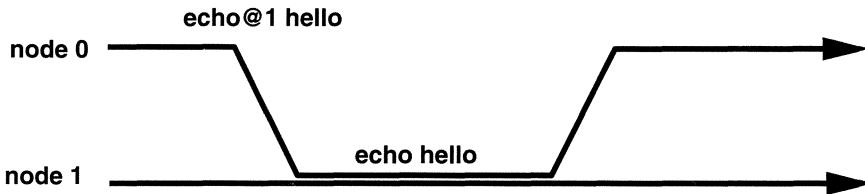


Figure 21.1 Synchronous Remote Function Call. Node 0 issues the command “`echo hello`” to node 1, and suspends. When node 1 has executed the command and returned the result to node 0, node 0 continues.

21.4.9 Asynchronous Remote Function Call

It is also possible for a node to issue a remote function call asynchronously using the *async* command. We recommend that only experienced users of PGENESIS use this command. A simple example is:

```
async echo@1 hello
```

In this case, the issuing node sends off the request for the command to be done to the executing node (1 in this example) and continues processing its script without waiting for the executing node to complete (or even start) the command. This is shown schematically in Fig. 21.2. Notice that after the remote function call has been issued by node 0, and until the result is received, there are three separate threads of execution: the parent on node 0 which continues; the child on node 1 which executes concurrently with the parent; and the original thread on node 1. Every use of the *async* command introduces one or more additional threads of control beyond the initial single thread per node with which PGENESIS starts.

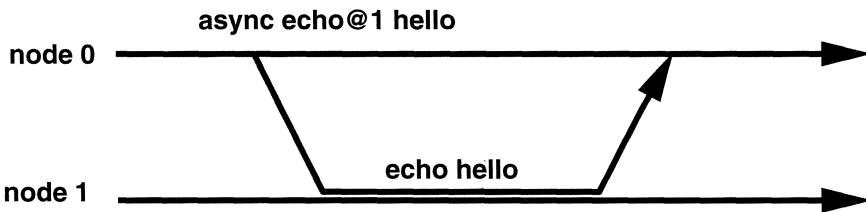


Figure 21.2 Asynchronous Remote Function Call. Node 0 issues the command “echo hello” to node 1, and continues executing. When node 1 has executed the command it returns the result to node 0, which does not use it.

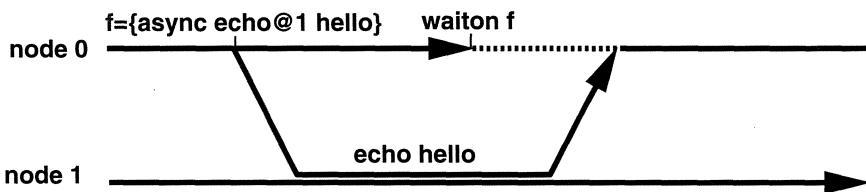


Figure 21.3 Completing an Asynchronous Operation. “echo hello” to node 1, and continues executing. When node 1 has executed the command it returns the result to node 0, which does not use it.

Every asynchronous operation issued from a node returns a result to the issuing node upon completion. A script can issue an asynchronous command, do some further processing, and then wait for the command to complete:

```
int future
future = {async some-function@1 args... }
some-useful-function args...
waiton future
```

The execution diagram for this script fragment is shown in Fig. 21.3.

The *future* variable is a handle returned by the *async* command that is passed to the *waiton* command. The script then suspends until the asynchronous operation completes (the future is satisfied). This “join” operation results in a reduction in the number of threads of control. The number of threads that terminate is exactly the same as the number of threads that were created by issuing the asynchronous operation.

A special form of the *waiton* command allows a node to wait for all its outstanding asynchronous operations to complete:

```
waiton all
```

This form of the *waiton* command reduces the number of threads of control that originated on the current node to exactly one. If every node executes this command followed by a barrier, then there will be exactly one thread of control per node when the barrier is satisfied.

21.4.10 Message Creation

For connecting elements that reside on the same node, the usual GENESIS commands (*addmsg* and *volumeconnect*) are available. However, PGENESIS also supports the creation of messages between elements that reside on different nodes. *raddmsg* allows one to create a message between an element on the node where the *raddmsg* is executed (the “local” node) and an element on another node (the “remote” node). For example, to create a PLOT message from */cell/soma* on node 1 to a graph on node 0, the following should be executed on node 1.

```
raddmsg /cell/soma /data/voltage@0 PLOT Vm *volts *red
```

One can also create this message from node 0 by using the remote procedure call mechanism:

```
raddmsg@1 /cell/soma /data/voltage@0 PLOT Vm *volts *red
```

There is also an *rvolumeconnect* command (analogous to *volumeconnect*) that is illustrated in the network example in this chapter (Sec. 21.6). It is currently not possible to delete remote messages.

21.5 Running PGENESIS

To run PGENESIS, you should first make sure that it has been installed, following the directions in the distribution. If PGENESIS was not included with your GENESIS distribution, you may obtain it from the PGENESIS web site (Goddard and Hood 1996), along with the latest version of the documentation.

We start with one of the simplest PGENESIS scripts — a parallel version of the classic “hello, world” program. We’ll examine this script line by line, adding some line numbers that aren’t present in the actual script:

```
1 paron -nodes 3 -parallel
2 echo@0 hello from node {mynode}
3 barrier
4 paroff
5 quit
```

1. `paron -nodes 3 -parallel` tells PGENESIS to initialize the parallel capability running 3 nodes in total, in the synchronized form (i.e., all in a single zone). PGENESIS will start up two more nodes executing the same script.
2. `echo@0 hello from node {mynode}` tells the node to issue an echo command to node 0 which prints the *issuing* node number.

3. `barrier` causes all nodes to wait until the barrier is reached. The purpose of this barrier is to make node 0, which in this script is reporting which nodes are running, wait until all the nodes have had their `echo` command printed.
4. `paroff` causes each node to flush its standard output and standard error buffers (so that output is written out) and then enter a barrier. Thus, by the time this command completes, all nodes will have flushed their buffers.
5. `quit` exits to the UNIX prompt.

In summary, the effect of this script is that output similar to the following will appear on node 0.

```
hello from node 0
hello from node 2
hello from node 1
```

Because of the parallel nature of the code, the particular order of these messages is not deterministic, and so you may sometimes find, for example, that node 1's message appears first.

21.5.1 The *pgenesis* Startup Script

PGENESIS is usually run by executing the *pgenesis* script. This script performs some checks on the execution environment, starts the PVM daemon on the appropriate machines, and runs the appropriate initial executable for PGENESIS. To try this out, locate the *pgenesis* script and put it on your PATH. Henceforth we assume that typing “*pgenesis*” results in this script being run. Put the “hello, world” script listed above into a file, say, “*hello.g*”. Now you should be able to execute the “hello, world” script with:

```
pgenesis hello.g
```

The full set of flags for this script is described with the PGENESIS hypertext documentation (Goddard and Hood 1996), which is also included in the PGENESIS distribution. In addition to the usual flags available for GENESIS, some of those interpreted by PGENESIS include:

- config** *filename* The specified file should contain a list of hosts to use to run the scripts (e.g., “*axp0 axp1 axp2*”). Names should be separated by blanks or newlines.
- debug mode** Run the workers in their own separate windows to allow debugging at either the GENESIS script level or at the C code source level. Not all modes are supported on all platforms. If you specify an unsupported mode the *pgenesis* shell script will select an alternative. Valid modes are:

1. **tty** — run the workers in individual windows but not under any C debugger.
2. **dbx** — run the workers in individual windows under the control of dbx.
3. **gdb** — run the workers in individual windows under the control of gdb running inside emacs.

-nox Run a version of the PGENESIS executable that does not have the XODUS libraries loaded — this is smaller, starts up faster, and does not require you to be running X windows. Note this only applies to the first node. Subsequent nodes are started with the *paron* command in the script, which can specify an executable.

-v Run in verbose mode.

-help Print text describing all the flags.

21.5.2 Debug Modes

The tty debug mode supported by the *pgenesis* script can be useful for debugging parallel scripts. The other two debug modes, which run PGENESIS nodes under the dbx or gdb debuggers, are most useful for advanced users who have interfaced their own C code with GENESIS.

To illustrate the use of the tty mode, and some of the functionality provided by the PGENESIS extensions to the script language, we will show you how to do some interactive programming of PGENESIS. This is not the way to develop parallel scripts, but it will give some insight into PGENESIS, and it can be useful in debugging parallel scripts.

First create a script *debug.g* containing this single command, which tells PGENESIS to run with 3 nodes and with the silent level at 0, i.e., so that all the usual banner and error messages are printed:

```
paron -nodes 3 -silent 0 -parallel
```

Now run PGENESIS with:

```
pgenesis -debug tty debug.g
```

This should start up a node that spawns two new nodes, so that there are three PGENESIS nodes running. Each of the spawned nodes appears in its own window, and after startup all nodes show a prompt.

Execute a remote function call from any of the nodes:

```
echo@all hello from {mynode}
```

This causes a “hello” message to appear in each node window.

Now try a barrier. Type “`barrier`” to the prompt in each of the node windows. Notice that the prompt does not reappear in any window until the `barrier` command has been issued on each node.

Now exit. Type “`quit@all`” to any of the prompts. The spawned node windows will disappear and, after cleaning up, the original node will exit to the UNIX prompt.

21.6 Network Model Example

[If you are not interested in distributing a network model over multiple nodes, this section may be skipped.]

This section illustrates how the `Orient_tut` example in Chapter 18 can be parallelized. Recall that that network has an array of retinal cells whose axons make contact with two populations of V1 cells. In parallelizing a network model, the most critical decision is how to decompose the network, i.e., how to distribute the cells amongst PGENESIS nodes. The goal is to minimize the number of synapses crossing node boundaries while maximizing the axonal delay of those synapses that do cross node boundaries. In `Orient_tut`, the connectivity is feedforward from the retina to V1 with some spatial divergence. A simple but effective decomposition is to divide the x -dimension of the retina and V1 populations amongst processors, as shown in Fig. 21.4.

In explaining this example, we do not provide the full scripts. These reside in the `Scripts` directory of the PGENESIS distribution. Instead, we show the important features of the parallelization, particularly how the simulation is set up and controlled and what is modified from the serial version described in Chapter 18. Nor do we discuss the issues involved in generating an XODUS display, which are covered later in Sec. 21.8.

21.6.1 Setup

In setting up the simulation, we will use one node to control the simulation, and the remaining nodes to run the slices of the model. Thus, n slices will require $n + 1$ nodes. These will run in a single zone (i.e., stepping is synchronized), with node 0 controlling, and nodes 1, …, n (the workers) running the slices. The main global variables used are:

```
int n_slices = 5          // number of slices
int retina_nx = 10        // retina has 10 x 10 cells
int retina_ny = 10
int v1_nx = 5            // each V1 population has 5 x 5 cells
int v1_ny = 5
int sim_steps = 1000      // when sweep is requested, we will do 100 steps
int i_am_control_node, i_am_worker_node // Booleans indicating the function
int slice                  // what slice this node holds
```

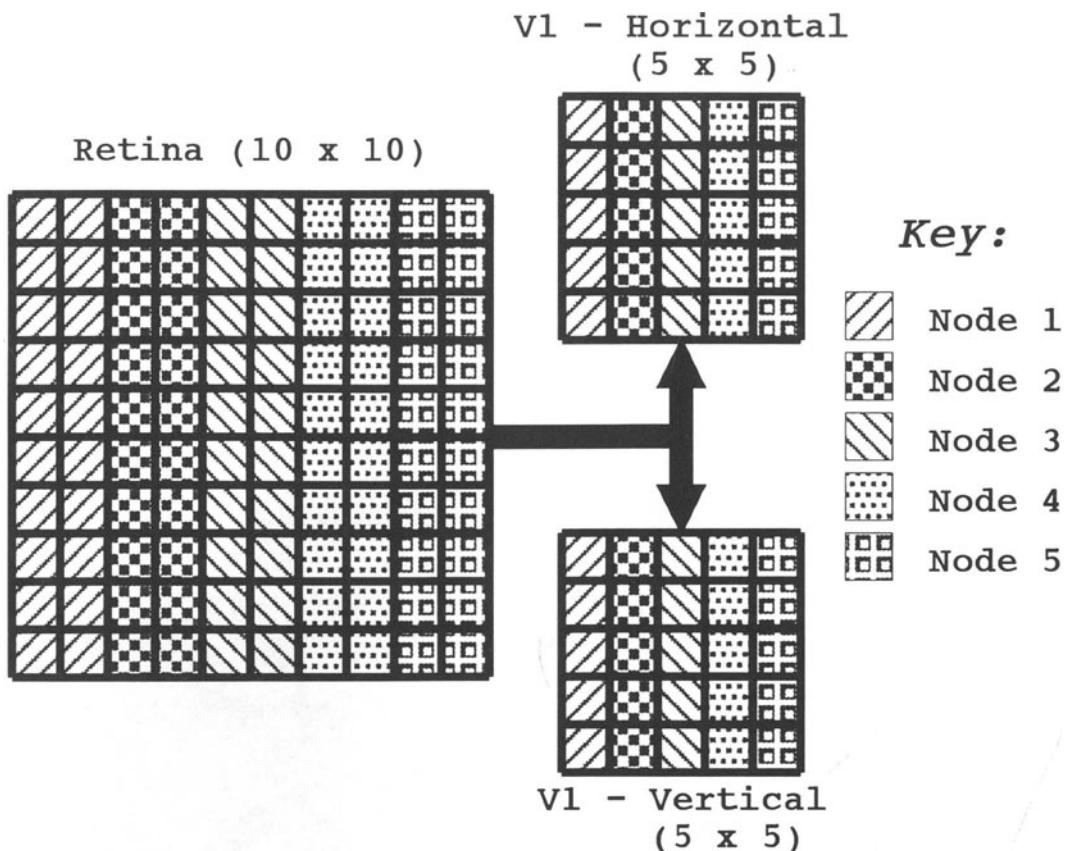


Figure 21.4 Slice decomposition of the retinal and V1 cells onto a set of 5 nodes.

```

str workers           // a list of nodes holding slices
int i                 // iterator
int n_nodes = n_slices + 1

```

The setup part of the controlling script, executed by all nodes, is:

```

1 workers = "1"
2 for (i=1; i<=n_slices; i=i+1); workers=workers @ "," @ {i}; end
3 paron -parallel -silent 0 -nodes {n_nodes}
4 setfield /post msg_hang_time 100000
5 i_am_control_node = {mynode} == 0
6 i_am_worker_node = {mynode} > 0
7 randseed {mynode * 347}
8 if (i_am_control_node); setup_control
9 else; create_slice_cells; end

```

```

10 barrier
11 if (i_am_worker_node); connect_retinal_slice; end
12 reset

```

Again, this listing has added line numbers that are not present in the actual script. This script creates a string containing the list of worker nodes (lines 1–2), then initializes PGENESIS (line 3) with n_nodes nodes. Line 4 sets the timeout to a very large value so that worker nodes, which wait at a barrier (line 23) will not timeout. Lines 5–6 set Booleans indicating the function of the node. Recall that every node executes this script and *mynode* returns the number of the node. Line 7 initializes the random number generator with a different value for each node, to avoid identical random number sequences on the different nodes. If the node is the control node, *setup_control* is called to initialize the control process (line 8). If it is a worker node, the slice cells are created (line 9). The purpose of the barrier in line 10 is to ensure that all the cells have been created before any node attempts to make connections (line 11). Although the control node makes no cells, it must participate in the barrier because a barrier always includes all nodes in a zone. After connections have been made, all nodes reset (line 12). The remainder of this script fragment (lines 13–24) appears in the following section on simulation control.

The code to create the cells in slices is only marginally different from that for the serial version (in *Orient_tut/retina.g*). For example, the retinal slice is created with:

```

createmap /library/rec /retina/recplane {REC_NX / n_slices} {REC_NY} \
    -delta {REC_SEPX} {REC_SEPY} \
    -origin {-REC_NX * REC_SEPX / 2 + \
        slice * REC_SEPX * REC_NX / n_slices} {-REC_NY * REC_SEPY / 2}

```

Here, the globals *slice* and *n_slices* are used to determine how many cells to create and what their spatial locations are. As described in Sec. 18.6, the *createmap* commands perform calculations that create the V1 populations on two-dimensional grids.

The connections from retinal to V1 cells are made with calls to *rvolumeconnect*. This command is just like *volumeconnect* except the destination path indicates on which nodes to look for the destination elements. Here we specify all the slice nodes. Although we could compute exactly which nodes should have the appropriate destination cells, it is easier to just ask PGENESIS to check everywhere. However, this does result in unnecessary communication between nodes during setup. In this example it is not a significant factor, but in more complex examples, especially with many nodes, it could be much more efficient to do this computation in the script. The connections to the V1 horizontal cells, for example, are made thusly:

```

rvolumeconnect /retina/recplane/rec[]/input \
    /V1/horiz/soma[]/exc_syn@{workers} \

```

```
-relative                                \
-sourcemark box -1 -1 0  1 1 0      \
-destmask box {-2.4 * V1_SEPX} {-0.6 * V1_SEPY} {-5.0 * V1_SEPZ} \
{ 2.4 * V1_SEPX} { 0.6 * V1_SEPY} { 5.0 * V1_SEPZ}
```

After the connections have been established, we can modify the axonal delays and synaptic weights with *rvolumedelay* and *rvolumeweight* which are analogs of the *volumedelay* and *volumeweight* commands in GENESIS. For example:

```
rvolumedelay /retina/recplane/rec[]/input -radial {CABLE_VEL}
rvolumeweight /retina/recplane/rec[]/input -fixed 0.22
```

21.6.2 Simulation Control

Lines 13-24 of the main script fragment, continued from above contain the crucial code for controlling the simulation:

```
13 if (i_am_control_node)
14   if (batch)
15     autosweep horizontal
16     barrier 7
17     paroff; quit
18   else
19     echo issue commands, then quit@all to terminate
20   end
21 else
22   barrier 7 100000
23   paroff; quit
24 end
```

Worker nodes simply sit at a barrier (line 22), ID number 7 (chosen simply for uniqueness), waiting for commands from the control node for a maximum of 100,000 seconds. If the simulation is running interactively, the control node prints a message (line 19) and then the script terminates, returning to the PGENESIS prompt. If the script is running in batch mode, the control node executes a simulation (line 15), then satisfies the barrier at which the workers are waiting. This allows the workers to continue on and quit (line 23). The control node similarly quits (line 17).

The function *autosweep* calculates the parameters for sweeping a bar across the retina and then steps the simulation on all nodes for the appropriate number of steps, setting the appropriate input in the retina before each step.

```
1 function autosweep
```

```

2  init_bar_params
3  for (i = 0; i < sim_steps; i = i + 1)
4      compute_bar_corners
5      setfield@{workers} /retina/recplane/rec[x>{x1}] [y>{y1}] \
6          [x<{x2}] [y<{y2}]/input rate {rate} -empty_ok
7      step@all
8  end
9 end

```

init_bar_params (line 2) is a script function not shown here that initializes the computation of the sweeping bar. Prior to each simulation step (line 7), the corners of the bar are computed with *compute_bar_corners* (line 4), not described further. This sets the global variables *x1*, *y1*, *x2*, *y2*, which are used in wildcard tests in the *setfield* command (lines 5–6), which actually sets the input in the retinal cells. Recall that *autosweep* is only called on the control node (see line 15 at the beginning of this subsection). It issues the *setfield* command for each node, so that all slices of the retina are properly initialized for the step. Notice the added flag “*–empty_ok*”. This flag is an addition to the *setfield* command which tells it that it is *not* an error if no elements match the wildcard specification. In the sliced up simulation, some nodes may not contain any retinal cells that are inside the spatial extent of the bar. It is simpler to allow *setfield* to accept an empty wildcard list of elements than to compute, for each step, exactly which nodes have relevant retinal cells and which don’t.

21.6.3 Lookahead

In network models, axonal delays are typically one or two orders of magnitude greater than the simulation time step for processes within a single cell. A spike generated at simulation time *T* need not be delivered to a destination cell until time *T + L*, where *L* is the axonal delay. This allows simulation nodes to operate in a loosely synchronized fashion, some being ahead of others, and it allows nodes to continue updating their cells while incoming spikes are in transit over the physical medium (e.g., Ethernet) that connects the CPUs. The amount of simulation time by which node A can get ahead of node B and still be sure it has not missed any spikes is known as the *lookahead* of A with respect to B.

In PGENESIS lookahead is controlled with three commands: *setlookahead*, *getlookahead* and *showlookahead*. The lookahead of node A with respect to node B is the minimum delay on all data paths from B to A, i.e., the minimum axonal delay over all the connections from B to A. If there are no axonal paths from B to A, the lookahead is infinite because A’s activity does not depend on B’s. If there are non-spike messages, lookahead is *dt* because those messages deliver data on the next time step. In addition, PGENESIS must be instructed to execute steps asynchronously with the command:

```
setfield /post sync_before_step 0
```

`/post` is the element that provides access to PGENESIS configuration fields. The variable `sync_before_step` is a Boolean indicating whether PGENESIS should synchronize nodes in a zone before a simulation step. By default it has value 1, so that nodes synchronize. To use lookahead, synchronization before stepping must be turned off.

By default, the lookahead is set to the time step, since every PGENESIS node always delivers spikes on the next time step. To set the minimum lookahead of a node with respect to all other nodes to 10 msec:

```
setlookahead 0.01
```

To set the minimum lookahead of a node with respect to, e.g., node 3 to 10 msec:

```
setlookahead 3 0.01
```

To find the lookahead of this node with respect to, e.g., node 4:

```
getlookahead 4
```

To view the lookahead of this node with respect to all other nodes:

```
showlookahead
```

It is wise to partition a model in such a way as to maximize lookahead between every pair of nodes. Techniques for automated partitioning are under investigation, but intelligent choice of partition by the modeler will remain a critical aspect of efficient simulation for some time to come.

21.7 Parameter Search Examples

[If you are not interested in parameter search, this section may be skipped.]

As mentioned in Sec. 7.4.1, GENESIS contains objects and commands for performing automated parameter searches, in order to estimate a set of model parameters that gives the best fit of the model behavior to the results of experiments. To demonstrate the use of PGENESIS for parameter estimation, we have constructed an example that performs a simple genetic search. The search problem in this example is quite trivial — we are trying to find values of parameters a and b that minimize $\min(|a - 1| + |b - 1|, |a + 1| + |b + 1|)$. This function has two minima, one at $(-1, -1)$ and one at $(1, 1)$, so our search procedure should find one of these. A more realistic parameter search in the neuroscience domain would have parameters that represent, for example, channel conductance densities in a cell model. The evaluation of the parameter set would be obtained by first running a neural simulation for, e.g., 100 milliseconds, and then comparing the simulation results (e.g., voltages or spike

times) with experimental data, and generating a numerical value to represent the goodness of the match. However, we have substituted here a simple function evaluation so that we can more cleanly illustrate a method for doing this type of search using PGENESIS. We go through the entire script in this section, starting with the high-level design and gradually refining the implementation in script commands.

In performing the genetic search, we keep a fixed-sized population of individuals “alive.” We randomly pick an individual from this population and mutate its representation with a certain probability. We then evaluate the new individual and, if it is better than some other, we replace the worst-evaluated individual in the population with this new individual. To allow for parallelism, we evaluate multiple new individuals simultaneously, and only remove existing individuals in the population as new individuals with superior evaluations are found.

In this example, we represent each parameter (out of a total of 2 parameters per individual) as a 16-bit string. The floating point value that this bit-string represents is determined by linearly mapping the 16-bit integer with range [0,65535] into the range [-32.768, 32.767]. When constructing a new individual from an old one, we mutate each of the bits with probability 0.02. We evaluate each individual by computing a trivial function over the parameters. In a real-world parameter search, this step would be the most time-consuming, since it would involve running a neural simulation. The main PGENESIS script follows, with each statement numbered.

```

1 paron -farm -silent 0 -nodes {n_nodes} -output o.out -executable nxgenesis
2 echo@0 node {mytotalnode} started
3 barrierall
4 if ({mytotalnode} == 0)
5     search
6 end
7 barrierall 7 1000000
8 paroff
9 quit

```

This is the top-level execution path for the nodes. The *paron* command (line 1) starts up the nodes in farm mode; i.e., each node is in its own zone, with output going to file *o.out*, and using the non-XODUS executable for spawned nodes since they do no graphical display. Line 2 uses a remote function call to print a startup message on node 0. Line 3 causes all nodes to synchronize here. *barrierall* is used because the nodes are in separate zones — recall that the *barrier* command synchronizes the nodes in a zone, whereas *barrierall* synchronizes all the nodes in every zone. Line 5 is only executed by the global zero node (*mytotalnode* gives the global node number, and *mynode* gives the node number within the zone). Line 5 is a call to the function *search*, described below, which conducts the parameter search. The non-zero nodes continue to line 7, which causes them all to wait at the barrier (with ID 7 and timeout 1 million seconds). The zero node does not reach this barrier until the call to *search* in line 5 returns, i.e., until the search is complete. At that point, the zero

node satisfies the barrier and all nodes flush their buffers and synchronize (line 8) then exit (line 9). In summary, the non-zero nodes sit at a barrier while the zero node conducts the search; then all nodes exit.

We use a number of global variables in the script:

```
int n_nodes = 4           // number of PGENESIS nodes to use
int individuals = 1000    // number of individuals to evaluate
int population = 100      // size of population to maintain
float bit_mutation_prob = 0.02 // mutation probability
float min_fitness, max_fitness // current worst/best fitness values
int actual_population = 0   // size of current population
int least_fit, most_fit    // indices into fitness data structure
int free_index = 0          // index in farm of free worker
int bs_a, bs_b              // bit string representation of parameters
```

While the non-zero nodes are sitting at a barrier, they can process incoming remote function calls and other requests from other nodes. The zero node conducts the search by issuing remote function calls to the non-zero nodes to evaluate individuals and return fitness values. The function that executes this search is:

```
1 function search
2   int i
3   init_search; init_farm
4   for (i = 0; i < individuals; i = i + 1)
5     if (i < population); init_individual
6     else; mutate_individual {rand 0 actual_population}; end
7     delegate_task {i} {bs_a} {bs_b}
8   end
9   finish; echo; echo "Finished search at " {getdate}; print_best
10 end
```

In this function, line 3 initializes data structures used in conducting the search (*init_search*) and managing the farming out of tasks to nodes (*init_farm*). The loop in lines 4–8 farms out the evaluations to the nodes. Lines 5 and 6 select parameter values for the evaluation. An initial population is selected randomly (*init_individual*), after which new individuals are derived from the existing population by mutation (*mutate_individual*). Line 7 sends the task to the worker (*delegate_task*) using parameter values *bs_a* and *bs_b*. Line 9 ensures the search has completed (*finish*) and then prints the best match (*print_best*). These functions are described below.

```
function init_search
  create neutral /gs; disable /gs
  create neutral /gs/population
```

```

addfield /gs/population a_value; addfield /gs/population b_value
addfield /gs/population fitness
createmap /gs/population / {population} 1
end

function init_farm
    create neutral /farm; disable /farm
    create neutral /farm/free; addfield /farm/free value
    createmap /farm/free / {n_nodes-1} 1
    for (i=0; i<{n_nodes-1}; i=i+1); setfield /free[{i}] value {i+1}; end
    free_index = n_nodes - 2;
end

```

These two functions create (with *createmap*) and initialize the data structures used to control the search (*/population*) and the farming out of tasks to nodes (*/free*). Each individual in the */population* vector has fields for the values of the two parameters and the fitness those parameters provide. The entries in the *free* vector have one field, the zone number of a node that does not currently have a task assigned. The *free_index* variable is an index into this vector of the last entry that contains a free node.

```

function init_individual
    bs_a = {rand 0 65536}; bs_b = {rand 0 65536}
end

function mutate (v)
    int v, i, b = 1
    for (i = 0; i < 16; i = i + 1)
        if ({rand 0 1} < bit_mutation_prob); v = v ^ b; end
        b = b + b
    end
    return {v}
end

function mutate_individual (chosen)
    int chosen
    bs_a = {mutate {getfield /population[{chosen}] a_value}}
    bs_b = {mutate {getfield /population[{chosen}] b_value}}
end

```

The three functions above generate a new individual. *init_individual* generates random parameter values. *mutate_individual* generates a parameter values by mutating the *chosen* individual with *mutate*.

```
1 function delegate_task
```

```

2   while (1)
3     if (free_index >= 0)
4       async worker@0.{getfield /free[{free_index}] value} {bs_a} {bs_b}
5       free_index = free_index - 1;
6       return
7     else; clearthreads; end
8   end
9 end

function finish
  while (free_index < n_nodes - 2); clearthreads; end
end

```

These two functions control the assignment of tasks to nodes and the gathering of results. *delegate_task* waits until there is a free node, then issues an asynchronous remote function call to the worker to evaluate the current parameterization (*bs_a*, *bs_b*). Line 3 tests for a free node. If there is one, the task is assigned (line 4), the node is removed from the free set (line 5), and the function returns (line 6). If there is no free worker, worker responses are checked for (line 7) and the loop iterates (line 2) until one is found.

This function prints out the best match found during the search:

```

function print_best
  float a, b
  a = ({getfield /population[{most_fit}] a_value} - 32768.0) / 1000.0;
  b = ({getfield /population[{most_fit}] b_value} - 32768.0) / 1000.0;
  echo "Best match with a = " {a} ", b = " {b} ", fitness = " {max_fitness}
end

```

There is one other function which executes on the node that controls the search (zone 0). When a node has completed an evaluation, it issues a remote function call for the controlling node to report the result. It calls:

```

1 function return_result (node, bs_a, bs_b, fit)
2   int node, bs_a, bs_b
3   float fit
4   if (actual_population < population)
5     least_fit = actual_population; min_fitness = -1e+10;
6     actual_population = actual_population + 1
7   end
8   if (fit > min_fitness)
9     setfield /population[{least_fit}] fitness {fit}
10    setfield /population[{least_fit}] a_value {bs_a}
11    setfield /population[{least_fit}] b_value {bs_b}
12    if (actual_population == population); recompute_fitness_extremes; end
13  end

```

```

14 echo " " {fit} -n
15 free_index = free_index + 1
16 setfield /free[{free_index}] value {node}
17 end

```

return_result adds the individual (lines 9–12) to the population if either there is room (line 4), or if the individual has fitness greater than some individual currently in the population (line 8). It prints the fitness (line 14) and puts the node in the */free* vector (lines 15–16). Notice in line 12 that if the population is full, then the individuals with minimum and maximum fitness are computed with *recompute_fitness_extremes*, shown below.

```

function recompute_fitness_extremes
    min_fitness = 1e+10; max_fitness = -1e+10
    for (i = 0; i < actual_population; i = i + 1)
        if ({getfield /population[{i}] fitness} < min_fitness)
            least_fit = i; min_fitness = {getfield /population[{i}] fitness}
        end
        if ({getfield /population[{i}] fitness} > max_fitness)
            most_fit = i; max_fitness = {getfield /population[{i}] fitness}
        end
    end
end

```

Since this function is called by *return_result*, it also executes on node 0, which controls the search. The evaluating node code is much simpler in this example because our evaluation function *evaluate* is trivial. In a real application, the evaluation would be computed by running a GENESIS model and comparing its output with experimental data. The two functions that execute on the evaluating node in our example are:

```

function worker (bs_a, bs_b)
    int bs_a, bs_b
    float a, b
    float fit
    a = (bs_a - 32768.0) / 1000.0; b = (bs_b - 32768.0) / 1000.0;
    fit = {evaluate {a} {b}}
    return_result@0.0 {mytotalnode} {bs_a} {bs_b} {fit}
end

function evaluate (a, b)
    float a, b, match, fit
    match = {min {{abs {a-1}} + {abs {b-1}}} {{abs {a+1}} + {abs {b+1}}}}
    if (match != 0.0); fit = 1.0/{sqrt {match}}
    else; fit = 1e+9; end

```

```

    return {fit}
end

```

worker is the function called by the controlling node to execute an evaluation. It converts the bit string representation of the parameters to floating point values, computes the fitness with a call to *evaluate*, and returns the result to the controlling node (0.0) using a remote function call. In our example, *evaluate* computes a match value that has two minima at $(1, 1)$ and $(-1, -1)$, and returns a fitness value that is the inverse of the square root of the match.

21.8 I/O Issues

Parallel simulations can often benefit from visualizations with XODUS during development, and often will require large input and/or output files in full scale production runs. Modelers should be aware that the way these I/O issues are dealt with can have a considerable impact on performance.

PGENESIS includes a capability to allow multiple nodes to display on the same XODUS widget so that, for example, a single **xview** can be used to show activation of all the cells in a distributed V1 layer. In serial GENESIS there are several ways to set up input to an **xview** element, described in Chapter 18 and in the documentation for **xview** in the GENESIS Reference Manual. However, the one we must use for internode communication in PGENESIS is to set up remote messages from a source element to a destination **xview** element. This is done by using the PGENESIS *raddmsg* command.

If every node were to set up COORDS and VAL n messages independently, the VAL messages could easily get associated with the wrong COORDS messages, depending on the order in which the particular add message requests were handled. To deal with this difficulty, the standard GENESIS **xview** object has been extended in PGENESIS to allow IVAL n messages to be associated with a particular ICOORDS message. The user does this by choosing an integral index with each message that is set up, and passing it as the first parameter of the ICOORDS and IVAL n messages. IVAL1 through IVAL5 messages will be associated with ICOORDS messages having the same index. For an example of this, see the example script *Scripts/par_io/par_view.g* in the PGENESIS distribution.

PGENESIS also includes a capability for writing a single disk file from multiple nodes. For disk output in serial GENESIS, it is typical to create an **asc_file** element and then set up a SAVE message that will cause a value to be written to a file on every time step. In PGENESIS it is possible to add such messages from elements on various nodes. However, there is no guarantee of order for the normal **asc_file** object, so in PGENESIS the **par_asc_file** object is provided. When SAVE messages are set up, the first parameter is an integral index that is used to maintain a fixed ordering among all of the various incoming messages to the **par_asc_file** element. This integral index should be unique and in the range from 0 to

the number of incoming messages minus 1, inclusive. Serial GENESIS uses the **disk_out** object for writing array data to a file in an efficient binary format. We have similarly provided a corresponding **par_disk_out** object that takes an added integral index parameter. The *Scripts/par_io/par_out.g* file in the PGENESIS distribution illustrates the use of this object.

Both of these extensions for I/O support require that information flow in the form of PGENESIS messages from the source node to the destination node (which holds the **xview**, **par_asc_file**, or **par_disk_out** element). If you are doing very large amounts of I/O from many nodes, the destination node would likely become a simulation bottleneck. In those situations, it would likely be advantageous to consider a solution where each node was doing its I/O to and from files on the local disk, rather than using the above mechanisms.

21.9 Summary of Script Language Extensions

21.9.1 Startup/Shutdown

To use any of the capabilities of the parallel library, one must first start up the library. This will also spawn the requested number of worker nodes on architectures that support process-spawning.

paron Starts up the parallel library.

paroff Shuts down the parallel library.

There are several commands for obtaining configuration information:

mynode Number of this node in this zone.

nnodes Number of nodes in this zone.

myzone Number of this node's zone.

nzones Number of zones.

ntotalnodes Number of nodes in all zones.

mytotalnode Unique number over all zones for this node.

mypvmid Task identifier used by PVM for this node.

npvmcpu Number of CPUs used by PVM in the parallel machine.

The ability to run parallel threads can be turned on or off (the default is on) with the related commands:

threadson Re-enables parallelism.

threadsoff Disables parallelism.

clearthreads Process all queued requests from other nodes.

clearthread Process one queued request from another node.

21.9.2 Adding Messages

It is possible to create arbitrary messages between elements on different nodes using the *raddmsg* command:

raddmsg Adds message between the listed source elements and the listed destination elements (which may be designated to be on other nodes by means of the “@” notation).

The following routine displays internode messages correctly (and suppresses the display of the postmaster messages used to implement the internode messages).

rshowmsg Shows the messages (intranode and internode) associated with a given element.

21.9.3 Synaptic Connections

There are several routines that allow one to set up multiple synaptic connections across nodes. They are analogs of the normal GENESIS routines for setting up synapses.

rvolumeconnect Connects one group of elements in a volume to another, using source and destination element lists and masks.

rvolumedelay Sets delays of a group of synapses receiving input from a list of presynaptic elements in a volume.

rvolumeweight Sets weights of a group of synapses receiving input from a list of presynaptic elements in a volume.

21.9.4 Remote Command Execution and Synchronization

<i>command@nodelist</i>	Executes command on specified nodes synchronously (i.e., does not return until remote commands have completed and returned result).
-------------------------	---

<i>async command@nodelist</i>	Executes command on specified nodes asynchronously (i.e., returns integer “future” without waiting for result).
-------------------------------	---

<i>waiton</i>	Wait for completion of a specified <i>async</i> command.
---------------	--

<i>barrier</i>	Wait for all nodes in my zone to reach this point.
----------------	--

<i>barrierall</i>	Wait for all nodes in all zones to reach this point.
-------------------	--

21.9.5 PGENESIS Objects

postmaster	One postmaster (<i>/post</i>) is created per node by the <i>paron</i> command to manage internode synchronization and communication.
-------------------	--

par_asc_file	Analogous to asc_file , except uses an ordering index.
---------------------	---

par_disk_out	Analogous to disk_out , except uses an ordering index.
---------------------	---

21.9.6 Modifiable PGENESIS Parameters

Several parameters of PGENESIS can be modified by the user by setting field values in the `/post` element. These fields, and their meanings, are:

1. `sync_before_step`. A Boolean indicating whether nodes in a zone synchronize before a simulation step. The default value is 1 (true). Asynchronous simulation is required for the lookahead optimization and may be faster even without lookahead.
2. `remote_info`. A Boolean indicating if information about messages between nodes should be kept for display with `rshowmsg`. This is an overhead that could be dispensed with for mature models. The default value is 1.
3. `perfmon`. A Boolean indicating if performance statistics should be gathered. This is a feature under development, explained in the PGENESIS documentation. The default value is 0.
4. `msg_hang_time`. A floating point value indicating how many seconds PGENESIS should wait before timing out on remote operations. The default value is 120.0 seconds. If debugging interactively, it is often useful to set this to a very large value so that one does not have to worry about the timing out of other nodes.
5. `pvm_hang_time`. A floating point value indicating how many seconds before timing out that PVM internal operations should wait. When a PGENESIS node is waiting for some message it expects, it will print a message followed by dots every `pvm_hang_time` seconds. The default value is 3.0 seconds.
6. `xupdate_period`. A floating point value indicating the number of seconds between a PGENESIS node's requests that X events be processed, when it is waiting for some expected message. High values can cause poor response from XODUS widgets. Low values can have an adverse impact on performance — but you shouldn't be using XODUS if you want performance. The default value is 0.01 seconds.

21.9.7 Unsupported and Dangerous Operations

It is extremely easy to reach deadlock in parallel programs; one way to reduce the chances of this is the frequent use of barriers and sparse use of asynchronous commands. However, barriers can be expensive to execute and can reduce parallelism, so they should be placed judiciously in scripts.

The serial GENESIS `stop` command should be used only with extreme care in zones containing more than one node. PGENESIS executes an implicit barrier before performing a simulation step. If any nodes enter the barrier, then all nodes must, otherwise deadlock will result. It is very difficult to satisfy this requirement when the `stop` command is issued.

Issuing *step* commands must be done with care. Since the *step* command executes an implicit barrier, failure to observe the following rule can result in deadlock. The two safe methods to issue step commands are:

1. *step* commands are issued exclusively locally (i.e., no use of the @ operator with *step*).
2. remote simulation *step* commands (e.g., *step@all*) are issued by at most one node in a zone.

21.10 Exercises

1. Modify the “hello, world” program in Sec. 21.5 to have node 0 print “hello, world” on both nodes 1 and 2. Use the *pgenesis* script in the “–debug tty” mode to observe that the output is correctly produced on the worker nodes. Now, change the *paron* statement to include “–output *workers.out*” so that their output is redirected into a file. When doing this, be sure to invoke *pgenesis* without the “–debug tty” flag or that will override the file redirection.
2. Create a very simple network model on two nodes using the neural model provided in *Scripts/simple*. Incorporate this by *include*’ing the *neuron.g* file in that directory. Write a script to create neuron A on node 0 and neuron B on node 1. Create a synapse from A to B using the *raddmsg* command. Verify that the model works by creating a display element on both node 0 and node 1 using the *create_display* function. Manually fire neuron A on node 0 and watch that the neuron B on node 1 fires in succession.
3. Partition the network model example of Sec. 21.6 so that all retinal nodes reside on node 1, and all V1 cells on node 2. Display the V1 horizontal and V1 vertical cells in two separate windows, each controlled by node 2.
4. Construct a “central pattern generator” by connecting five neurons in a ring fashion with each neuron making an excitatory synapse onto its clockwise neighbor. Place each neuron on its own node and set the axonal delay to 5 msec. Investigate the use of the lookahead feature to speed up the simulation. Optional: use the performance monitoring capabilities described in the PGENESIS documentation.
5. Using the neuron model and real spike data file supplied with the PGENESIS distribution in *Scripts/experiment*, modify the parameter search of Sec. 21.7 to use a realistic evaluation function that employs a least-squared error method for comparing the simulated spikes to the actual data. Use it to find the values of the conductances for

the fast Na current and the delayed-rectifier K current that best match the data. Hint: create an evaluator element that accepts SPIKE messages and in the action handler (set by *addaction*); compare the simulated spike times against the spike times of the actual data.

Chapter 22

Advanced XODUS Techniques: Simulation Visualization

UPINDER S. BHALLA

22.1 Introduction

One of the biggest hazards in developing a simulation is the pressure to make it user-friendly. If you ever make the mistake of making a simulation easy for people to play with and understand, they will suddenly discover gaping holes in your model design, and start to think up all sorts of “improvements” for you to make. An even more unpleasant situation can arise when you have sent off the final page proofs of your simulation paper, and decide that now is a good time to provide it with a colorful display so that people reading your paper can run the simulation themselves. Inevitably, the display will reveal a fundamental bug in the simulation that no one (least of all yourself) would ever have noticed in all the hundreds of lines of simulation code. The prudent builder of simulations will avoid any compromises when it comes to obfuscation. This chapter, then, is for the reckless, since its stated goal is to reveal all, to shine the bright light of day on the hidden crannies of simulations where bugs lurk, and to display the gory details using the rainbow colorscale in an animated three-dimensional draw widget.

22.2 What Can Your User Interface Do for You?

A user interface is a tool for communication. It has a role wherever you wish to interact with the computer, with data, or with other people. Teaching, demonstrating simulations, or facilitating development are all common applications. A graphical interface can be much more than a set of buttons telling a simulation to start and stop, with a graph or two thrown in. You have already met examples of some of the more interesting things one can do with XODUS in the form of the demos such as *Orient_tut* (Chapter 17) and simulation building tools such as *Neurokit* (Chapter 7). In the technical sense, though, there are four main operations for which user interfaces, and XODUS in particular, are designed.

1. **Input.** An interface should make it easier to provide control signals for a simulation, and to assign parameters.
2. **Output.** An interface should simplify the monitoring of interesting aspects of simulation, and especially be able to provide an easily interpreted view of a complex simulation.
3. **Checking for errors.** This includes, but goes beyond, simply monitoring the progress of a simulation to see if it is behaving strangely. Several of the advanced XODUS widgets are designed to help with analyzing the structure of a simulation, so as to check that the simulation is connected up the way you think it is.
4. **Building simulations.** Writing a tool to help users build simulations is about the hardest user interface task. It embodies all of the aspects listed above, and has special challenges all its own.

In the course of working through this chapter you will be introduced to graphical components that are used for carrying out all the above user interface operations. The latter part of the chapter consists of an extended example that uses many of the user interface components to put together a simulation builder. Along the way you will get to see some of the organizing principles of XODUS, and how they all fit into the framework of GENESIS objects, actions, and functions.

22.3 Draw/Pix Philosophy

As long as one is restricted to displaying single data points (dialogs) or arranging widgets on the screen (forms), the graphical interface world is simple. Displaying anything more complex than text or bitmapped images brings in all sorts of complications such as scaling, rotation, managing events, and so on. In XODUS, all these more complex displays are handled by an elite family of widgets called *draw* widgets, and their children, which are

pix widgets.¹ Even such an apparently simple operation as plotting a graph turns out to be sufficiently complex that it is handled by a specialized version of a draw widget, although most of the complexities are hidden from you.

From the point of view of the user, a draw widget is a window onto three-dimensional space, and a pix is any item visible in that space. Depending on the perspective of the window, a pix may not be visible (for example, it may be off to the side of the window), or it may be occluded (another pix may be in front of it) or it may simply be too small to see (a 5-micron neuron is not easy to find in a window representing a volume of space that is a meter on each side). There is the inevitable tradeoff here: the flexibility in being able to look at three-dimensional objects and do zooms, pans and rotations has to be paid for by specifying more parameters and by having to do more display computations.

There are some parallels between the draw/pix relationship and the familiar form/widget relationship. First, a pix can only be displayed in a draw. Second, in order to be displayed by a draw, the pix must be a child (or descendant) of the draw. Third, the draw “manages” the pix. Just as a widget is at the mercy of the parent form with regard to resizing, hiding and so on, a pix is only displayed according to what the parent draw decides. A basic organizing principle is that any pix can be displayed in any draw. There is a common set of properties for all draws, and likewise a common set of properties for all pixes, which make this work uniformly. To put it in computer science terms, all draws are subclassed from the *coredraw* object, and all pixes are subclassed from the *pix* object. The operations performed by the *coredraw* object and its subclasses include:

1. Performing coordinate transforms for the pixes. Different draws project pixes in different ways according to the transformations available in the draw widget.
2. Managing events for the pixes. These include all interface events such as mouse clicks, drags and drops, and resize requests. For each mouse event the draw widget must identify the destination pix and pass the event on to it.

Similarly, all pix subclasses carry out the following set of common operations.

1. Managing a set of coordinates for the display.
2. Doing the actual graphical display once the transformations have been completed by the parent draw widget.
3. Dealing with interface events that have been forwarded from the parent draw widget.

¹If one is really particular about nomenclature, one should refer to pixes as *gadgets* rather than *widgets*, since *gadget* is what the gurus at Project Athena have chosen to call windowless widgets. I do not bother with this distinction.

22.4 Meet the Cast

The forms, buttons and dialogs you have met in earlier chapters are the lower invertebrates on the interface evolutionary tree. The graph widget is actually a highly evolved interface component that has taken to slumming with the “simple widgets.” There is even a rumor that it underwent a frontal lobotomy in order to fit in better — brain damage, by any other name. You have had glimpses of some of the advanced widgets in the tutorials such as *Orient.tut* and *Neurokit*. In this section we provide brief sketches of their function and a simple example or two. As always, for detailed information, read the manuals and documentation, and look at the examples.

A note on nomenclature: all XODUS widgets and almost all XODUS-related commands begin with an “x”. This preceding “x” is usually omitted in the text, although all the examples have the commands and names in full.

22.4.1 The Draw Widget Family

Coredraw

This is the base class of all draw widgets. The only transformation it knows about is projection in the x - y plane. It is useful if the pix being displayed is flat and has no business using another projection.

Graph

This is a highly specialized draw widget, whose function you already know. Its projections are also confined to the x - y plane. It is designed specifically to manage axes and plot pixels, and does so in a very stereotypical and bossy manner, even interfering with their creation and destruction. As you have already seen, it creates child plots automatically when a PLOT message is sent to the graph widget. However, one can explicitly add any pix (including plots and axes) to the graph and do the usual manipulations on them.

Dumbdraw

This exists only as an example for those wishing to understand the C-code implementation of the draw widget hierarchy, and how to manage inheritance from the coredraw widget. It only differs from the coredraw in being able to perform projections in the x - z and y - z planes as well as the x - y plane. If one really needs the x - z and y - z planes, one could just as well use the draw widget.

Draw

This is the most general draw widget, providing general orthographic as well as perspective transformations. In the examples that follow we use this as illustrative of all the other draw classes, although they cannot handle all of the transformations that this does.

1. Create a draw widget, specifying the region of space that it displays and a function to execute.

```
create xform /form
create xdraw /form/draw [0,0,100%,100%] \
    -xmin -5 -xmax 5 -ymin -5 -ymax 5 -zmin -5 -zmax 5 \
    -script "echo Hello from <w>"
ce /form/draw
xshow /form
```

2. In order to see how the coordinate transformations work in the draw widget, we need to have something to display in it. We will use a simple **xshape**, which is a generic pix for drawing shapes, to draw an open green rectangle:

```
create xshape shape -fg green \
    -script "echo Hello from <w>" \
    -coords [0,0,0] [0,2,1] [3,2,1] [3,0,0] [0,0,0]
```

3. Let us first investigate the keyboard controls of the draw widget for zoom and pan. In order to use keyboard controls for manipulating a draw widget, the mouse must be positioned on the widget.

- Move the mouse onto the draw widget.
- Press the arrow keys on your keyboard. The rectangle should move around on the draw widget in the appropriate directions. These are the pan controls.
- Press the angle-brackets keys (i.e., the comma and period keys). These should shrink and expand the rectangle, respectively.
- Return the rectangle to a reasonable size and position for the next step.

4. Now let us look at the transformations provided by the draw widget.

- You are currently in the *x*-*y* plane, looking down from the *z* axis. So, the current transformation is identified by the “*z*” key. Hit the “*y*” key to go into the *x*-*z* plane. The rectangle will probably jump a little, because of the pan operations you did earlier. Its dimensions will also change, because you are now looking at it from another direction.

- Go into the y - z plane by hitting the “ x ” key. Now the rectangle turns into a tilted line. This is because you are now looking at it edge-on.
 - To get a better feel for the rectangle’s orientation in three-dimensional space, we will now go into an orthographic projection. This lets us look at the object from any direction, but without using perspective. Hit the “ o ” key to go into orthographic mode. The rectangle will jump again, and change shape.
 - As before, the arrow and angle bracket keys can be used to manipulate the pan and zoom of the display. Now, however, the pan operations can affect all three axes.
 - In the orthographic mode, we can change the viewpoint of the draw widget. The viewpoint is a three-dimensional vector along which the observer looks at the draw widget, and is defined by the fields vx , vy and vz . It is equivalent to the normal to the plane of projection of the draw widget. To rotate the rectangle about the z -axis, keep the “shift” key pressed while you press the left and right arrow keys. Check how the vx and vy fields change when you do this, while the vz field stays the same. Conversely, you can also assign values to the vx , vy and vz fields to set up predefined viewpoints from scripts.
 - To rotate about the horizontal, keep the “shift” key pressed while you hit the up and down arrows.
 - It is very likely that by now you have experienced the Necker cube illusion with the rectangle — it is hard to decide which end of it is near you, and which is further away. This is a drawback of the orthographic projection. So let us go the whole hog, and try the perspective projection. Hit the “ p ” key.
 - The rectangle should change shape slightly, and will also shrink by a factor of 2 or so. Try the rotation operations again. Can you see any effect of perspective?
 - The perspective transformation introduces yet another parameter: the distance of the observer from the object. As you get closer to the object, the amount of distortion introduced by perspective increases. If you get too near the object, really strange things might happen as error-trapping code kicks in. Play around with the perspective distortion by hitting the square bracket keys on the keyboard, i.e., “[” and “]”, and then watching the effect of rotating the viewpoint.
 - Note that in the limit, as one gets further away from the object, the perspective transformation approaches the orthographic projection (except for the scaling factor). Test this by toggling between the “ o ” and “ p ” projections.
5. Finally, let’s check out the effect of mouse actions on the draw widget and its contents. Events in the draw widget are directed to the nearest pix, if it is within a certain distance

(usually around 10 pixels) of the mouse event. Otherwise the draw widget handles the events itself.

- Click on a line of the rectangle. The rectangle should be transiently highlighted, and a line should appear on the console window saying: Hello from /form/draw/shape.
- If you click on a clear area of the draw widget, the event gets directed to the draw. The console now says: Hello from /form/draw.

We go into considerably more detail on events in XODUS in a later section of this chapter. Now we proceed to the members of the pix family of widgets.

22.4.2 The Pix Family

pix

This is the base class of all pix widgets, and its only purpose is to act as the family patriarch. If one creates a pix, it will draw a set of cross-hairs, which are not particularly useful, but we demonstrate it anyway. Continuing with the previous example of the draw widget:

```
create xpix pix
```

will put a pix widget in the draw. The pix widget looks the same, no matter what transformation you use. Try them out. When you have convinced yourself, go into the *x-y* plane (the “*z*” key) for the next set of examples.

The pix widget, and all its subclasses, have the following common set of fields:

fg – foreground color. Try it out by setting *fg* to various colors:

```
setfield pix fg yellow  
setfield pix fg blue
```

If you load a colorscale, then 64 colors are accessible as numbers from 0 to 63:

```
xcolorscale rainbow  
setfield pix fg 0  
setfield pix fg 32  
setfield pix fg 63
```

tx, ty, tz – translations to be applied to the pix as a whole, to move it around in space. For example:

```
setfield pix tx -1
```

will move the pix by 1 unit in the negative *x* direction.

script – script command(s) to execute in response to interface events. Assign a command to the pix:

```
setfield pix script "echo this is a pix widget"
```

Now click at the intersection of the crossbars to see what happens.

value – a text string with a value associated with the pix. This is often used as an argument for the script command, using the angle-bracket notation discussed in Sec. 22.5.1.

```
setfield pix value "12345"
setfield pix script "echo the value of the pix <w> is <v>"
```

Now click on the pix again.

pixflags – a set of flags that determine many of the properties of the pix including visibility, sensitivity to mouse events, how it handles transformations, highlighting, and so on. The *pixflags* command from the command line lists the options. A typical flag operation is to make the pix invisible:

```
setfield pix pixflags v
```

Then you can flip the flag back:

```
setfield pix pixflags ~v
```

Another common flag is used to turn off the sensitivity to the mouse:

```
setfield pix pixflags c
```

Now it will ignore mouse events.

sphere

This draws a circle. Its main purpose is to serve as a coding example for those wishing to create their own pixes.

```
create xsphere sphere -fg red -r 0.5
```

gif

This loads in a gif-format image file and displays it within the draw widget. It does not attempt to scale the image according to the transformations of the draw widget, but it does reposition the center of the image appropriately. Locate a suitable gif file. There should be one called *xodus.gif* in the *Scripts/examples/XODUS* directory.

```
create xgif gif -filename xodus.gif
```

If you cannot find a gif file, do not despair. The gif example is not needed for any of the following steps.

plot

This makes plots. It is created automatically by the graph widget when it receives a PLOT message, but can be created explicitly and is perfectly capable of handling messages on its own. It is a perfectly normal pix, so it can be created in other classes of draw widgets as well. It has all sorts of options relating to data compression, display modes, and so on, which are illustrated in previous examples and in the **xgraph** documentation and examples. There is a generic pixflags option that is especially relevant to the plot pix. This is the *flush* option, which turns off the forced updating of a pix after an update. When you have a dozen or so plots, it becomes pretty time-consuming to update the display a dozen times per time step.

axis

This makes axes in three dimensions. Two axis pixes are created automatically (for the *x* and *y* axes, respectively) when a graph widget is created. Again, the axis pix can be created explicitly as well, and is at home in any kind of draw widget. One can provide a vector for both the axis direction and for its tick marks, so it is not restricted to providing axes in the *x-y* plane.

shape

This is the workhorse pix. It does what its name implies, which is to draw shapes. The **xshape** pix is used extensively as an icon both on its own in a draw widget, and also as a subordinate pix for some of the really powerful pixes described below. We already have an example of the **xshape** up in the draw widget, which we will continue to manipulate here. First, we have to dig it out from underneath the gif widget.

```
setfield shape ty 3
```

One can display text in a shape widget:

```
setfield shape text "I am an xshape"
setfield shape textcolor blue
setfield shape textfont r24
```

The **xshape** has several drawmodes:

```
setfield shape drawmode FillPoly
setfield shape drawmode DrawSegments
setfield shape drawmode DrawPoints
setfield shape drawmode DrawArrows
setfield shape drawmode DrawLines
```

and so on.

The **xshape** manages a set of coordinates in the form of three *interpol_structs* (tables), for the *x*, *y* and *z* coordinates. These can be examined using the usual “*showfield -a*” command. Since the coordinates are stored in standard *interpol_structs*, one can use several commands for manipulating them. The most obvious is to simply set them:

```
setfield shape xpts->table[0] -1
```

There are many other options for manipulating *interpol_structs*, some of which are listed in Chapter 18. There is a special coordinate specification mode for **xshapes** which is retained for backwards compatibility and simplicity. It is meant to be used when the shape is being created, but you can also use it later:

```
create xshape newshape -coords [1,0,0] [0,1,0] [0,0,1]
setfield newshape coords [2,0,0] [0,1,0] [0,0,1]
```

var

This **pix** is used to display values graphically. It can vary almost any of the graphical parameters of **xshape** (e.g., color; coordinates; text *x*, *y* and *z* offsets) to represent changing values. This is quite a complex **pix**, and we honor it with a long example. The old example is pretty cluttered by now, so let’s clean it up a bit:

```
delete /form/draw
create xdraw /form/draw [0,0,100%,100%] \
-xmin -5 -xmax 5 -ymin -5 -ymax 5 -zmin -5 -zmax 5
ce /form/draw
```

The **xvar** assumes that a colorscale has been loaded. This was done several steps ago. If you haven’t been following through all the examples, now is the time to load in a colorscale.

```
xcolorscheme rainbow
```

Now create the new **xvar**:

```
create xvar var
```

The **xvar** displays values by interpolating between the relevant display parameters of two subordinate xshapes, which are created by default when the **xvar** is created.

```
genesis #25 > le var
shape[0-1]
```

Before we go any further, let's set up a couple of tables to use as inputs to the **xvar**.

```
create table /tab1 // this table will emit a triangle wave
call /tab1 TABCREATE 2 0 1
setfield /tab1 step_mode 1 stepsize 0.002 \
    table->table[0] 0 table->table[1] 1 table->table[2] 0
create table /tab2 // this table will emit a sawtooth wave
call /tab2 TABCREATE 2 0 1
setfield /tab2 step_mode 1 stepsize 0.001 \
    table->table[0] 0 table->table[1] 1 table->table[2] 2
addmsg /tab1 var VAL1 output
addmsg /tab2 var VAL2 output
```

First, let's look at the default display modes of the **xvar**.

```
reset
step 1000
```

You should see an expanding box, which changes color as it expands. This is the default display mode, called *colorboxview*. If you are the owner of an obscenely fast machine, this example may have whizzed by too fast to see. You can handicap your machine to get it to display at human speeds by reducing the step size of the tables by an order of magnitude, and increasing the number of steps correspondingly.

There are several other built-in display modes, which do fairly straightforward things to the display. Try:

```
setfield var varmode boxview
reset
step 1000
setfield var varmode colorview
reset
step 1000
setfield var varmode fillboxview
reset
step 1000
```

There are far more interesting things one can do with the var widget.

- At this point you are using the default square shapes that the var created. Let's change the coordinates of the first shape, so that instead of starting out as a small square, the var starts out as a big triangle:

```
setfield var/shape[0] coords [-2,0,0] [0,2,0] [2,0,0] [0,0,0]
reset
step 1000
```

You should see a triangle “morphing” into a square. As already mentioned, the way that var displays values is by interpolating between display parameters whose extreme values are specified by the child **xshapes**. In the *fillboxview* mode we saw previously, the original shapes were a small and a large square, respectively. Now the shapes are a triangle and a square. Since the current display parameter refers to the coordinates, the net effect of interpolation is morphing.

- You can display a value using more than one graphical parameter at a time. Let us use the y offset as another such parameter. First we need to have different y offsets in the shapes:

```
setfield var/shape[1] ty 3
```

Now we need to tell the var that it should use the information coming in on message VAL1 to display using the y offset:

```
setfield var yoffset_val 1
reset
step 1000
```

and you see that in addition to changing shape, the display is also bouncing up and down.

- You can display more than one value at a time. We already have two messages coming in to the var. Let's display the second one using colors from 0 to 63.

```
setfield var/shape[0] fg 63
setfield var/shape[1] fg 0
```

Note that the range of */tab2* is from 0 to 2. The minimum value is the same as the default, but we have to specify the appropriate upper limit in the *value_max* table.

```

setfield var color_val 2 value_max[1] 2
reset
step 1000

```

Many of these concepts of displaying values in terms of different graphical parameters are used in other pixes as well, such as the **xview** pix and the **xcell**.

view

As a first approximation, this pix is something like an array of **xvars**, but much more efficient. Suppose, in the previous example, you wanted to display 100 versions of */tab1*. If you were to make 100 **xvars**, that would mean 200 child **xshapes**, and each of them would have to be set up independently. Instead, you could use an **xview**. This would require just 2 **xshapes**, but the display would look just the same.

When an **xview** displays the values of many elements it usually arranges all the element icons on the screen based on the three-dimensional coordinates of the elements. These coordinates are specified automatically when using the *path* field option illustrated below, or using the COORD message when using the message-based display options.

An **xview** has a superset of the options available to **xvar**. One of the most important enhancements is the ability to use “paths” as well as messages for sending values to the **xview**. The *path* field specifies a wildcard list of elements that will send values to the **xview**. There are several other options related to the path field, which enable display of subelements from the path, of messages connected to the path, and so on. These are illustrated in the *Orient.tut* tutorial (Chapter 18), and discussed in some detail in the reference manual.

The **xview** happens to be ideal for building a very useful display feature, which I illustrate here:

```

create xform /scale [200,200,400,100]
create xdraw /scale/draw [0,0,100%,100%] \
    -xmin -2 -xmax 65 -ymin -2 -ymax 2
xcolorscale rainbow
createmap neutral /colorscale 63 1 -object
create xview /scale/draw/view -viewmode colorview \
    -path /colorscale/proto[] -field x -value_max[0] 64
reset
step
xshow /scale

```

Voila! you have just constructed a colorscale. Try loading in other colorscales to see what they look like:

```

xcolorscale hot
step

```

cell

The **xcell** pix, like the **xview** pix, is dedicated to displaying a lot of values at the same time. It is highly specialized for displaying cells, that is, compartmental models of neurons, where each compartment is assigned a diameter and position in three-dimensional space. The reader is referred to the reference manual and examples for further information.

tree

This pix is used to display, edit, and build simulations in XODUS. Typical uses include exploring and manipulating the element hierarchy, managing a library of objects, and building simulations using drag-drop operations. The **xtree** is related somewhat to the **xview** widget, in that it uses a set of subordinate xshapes to represent the simulation components it is displaying. Typically, each class of object is assigned a distinct **xshape** icon. There are several aspects to what **xtree** does:

1. It displays objects or elements graphically. The positioning can be in a tree hierarchy, or based on the three-dimensional coordinates of the elements, or in a grid, or user-defined.
2. It displays messages between elements.
3. It manages graphical interface events, especially drag-drop operations, between elements it displays. In particular, it enables the user to attach script functions to drag-drops between subsets of elements in the display.

Section 22.6 gives an extended example of a network builder using the **xtree** pix.

22.5 XODUS Events

At various points, we have mysteriously mentioned events as something to which XODUS objects respond. To be more specific, events are any user operations that impinge upon the user interface. In XODUS, all events are mapped onto standard GENESIS actions. Not all widgets or pixes recognize all events. Some of them (such as labels and forms) do not recognize any events at all.

All widgets that are sensitive to events have a *script* field (and a *-script* option used with *create*) which, as described in previous chapters, executes functions when something happens to the widget. So far, you have mainly used the default event, a standard mouse click. In order to select a specific event, one can attach a suffix to the end of the script function name. For example, we can set the script for the draw widget to respond to a click of mouse button #2 (the middle button):

```
setfield /form/draw script "echo.d2 This was mouse button 2"
```

Or, we could have it respond to a double click on any mouse button:

```
setfield /form/draw script "echo.D This was a double click"
```

We can specify multiple functions in the script field, and each can be associated with any event. Each such function is separated by a semicolon:

```
setfield /form/draw script \
"echo.d1 d1; echo.d1 Also d1; echo.d2 d2; reset.d3"
```

Table 22.1 shows the mapping of events to widgets.

	<i>Event</i>	<i>Action</i>	<i>Suffix</i>	<i>Widgets</i>
1	Mouse button click	B1DOWN etc.	none; or d, d1, d2, d3	all
2	Return to off state	B1UP etc.	u, u1, u2, or u3	toggle
3	Mouse double click	B1DOUBLE etc.	D, D1, D2, or D3	all
4	Update (internal)	XUPDATE	Not available	all
5	Keypress	KEYPRESS	k	dialog
6	Drag from (called from source widget)	XODRAG	y	draw/pix
7	Drop into (called from dest widget)	XODROP	p	draw/pix
8	Drop into (called from source widget)	XOWASDROPPED	w	draw/pix
9	Script access	XOCOMMAND	c	shape

Table 22.1 Mapping of XODUS events to actions.

22.5.1 Returning Arguments to Script Functions

A very powerful feature of the script-calling syntax is the ability to pass specific arguments to the function. These arguments are passed in using angle brackets. All widgets that have a script function can take the *<widget>* argument, usually abbreviated *<w>*. This returns the pathname of the widget that called them. In addition, all widgets that have a *value* field (or in the case of the toggle widget, a *state* field) can also pass the value using the *<value>* (or *<v>*) argument. The dialog widget can pass individual keypresses in the *<k>* argument. When one is doing drag-and-drop operations, one needs to keep track of the source and destination widget and their values. These are identified by the *<s>*, *<d>* and the *<S>*, *<D>* argument pairs, respectively. Finally, draws and pixels can all pass coordinate arguments: *<x>*, *<y>* and *<z>*. Most of these are illustrated in the network builder example that follows.

22.6 Using Advanced Widgets: A Network Builder

As a working example of many of the features we have discussed so far, we will build *Netkit*, a skeleton interface for constructing network simulations. At a first pass, you can see how the various XODUS components work together. At a more advanced level, particularly if you ever plan to build your own interface, it may be worthwhile to note how the interface has been modularized. This modular design has been developed for several recent simulation tools including *Neurokit2* and *Kinetikit*, and will enable you to merge your interface with these existing ones and make use of some of the powerful tools such as the parameter search genie. The network builder is designed to do the following operations.

1. Manage a set of prototype cells using an **xtree** element.
2. Drag prototype cells into an editor window.
3. Double-click on cells to edit them.
4. Click-and-drag between cells to set up connections.
5. Drag cells to a graph window to set up plots.

To avoid cluttering up the chapter with non-XODUS related code, the cell models have been relegated to Appendix B. Here we only go over the interface-specific portions of the network builder. If the prospect of typing in all this code (although it is under 150 lines) is intimidating, you may retrieve it from the *Scripts/examples* directory of the GENESIS distribution. The example presented here is organized in a stepwise progression suited to a tutorial. In the example files, and in real simulations, one would organize it much more systematically in the form of modules related to specific components of the interface.

22.6.1 The Library Window

To start off, we will set up the library window and the **xtree** that manages the prototypes. In this case, the **xtree** is doing two things of note. First, it is arranging the cell prototypes for you, by putting them into a grid, i.e., a rectangular array. These prototypes are identified here using the wildcard path, assuming that all objects of type **compartment** under the */proto* element are cells. Second, the **xtree** specifies a function call (*create_cell*, which we will write later) that will be invoked when the prototypes are dragged into the work window.

```
create xform /edit [0,150,500,550]
xshow /edit
create xcoredraw /edit/lib [0,0,100%,30%] \
```

```
-xmin -3 -xmax 3 -ymin -1 -ymax 2
create xtree /edit/lib/tree -path "/proto/##[TYPE=compartment]" \
-treemode grid \
-script "create_cell.w <d> <S> <x> <y>"
```

22.6.2 Making Prototype Cells

In order to give the library something to display, let us set up a couple of cell prototypes. First, the excitatory cell. Because there is a lot of uninteresting (at least from a graphical perspective) code involved in setting up a cell model, we will just load in the cell, using the *make_cell* function defined in the *cellproto.g* file. This puts it under the */proto* element which is always created by default.

```
include cellproto.g
make_cell // This loads a cell into /proto
```

Alternatively, if you can't find *cellproto.g* and are eager to proceed, just create a dummy cell by typing:

```
create compartment /proto/cell
create synchan /proto/cell/glu
create synchan /proto/cell/GABA
create spikegen /proto/cell/axon
```

This will not spike, of course, but for now it will get things started. Now we can set about turning this into an excitatory cell for our interface. First, we need to use the *move* command to rename the cell:

```
move /proto/cell /proto/eccell
```

Now we add some extended fields to the cell to handle user-interface information. The *transmitter* field, obviously enough, indicates what transmitter type the cell uses. The *xtree_fg_req* field is a special field name that the *xtree* widget recognizes, and is used to assign a color to the cell icon.

```
addfield /proto/eccell transmitter
addfield /proto/eccell xtree_fg_req
setfield /proto/eccell transmitter glu xtree_fg_req green
```

Well, that was easy enough. Now we make a copy for an inhibitory cell, which is just the same as the excitatory cell except for the color and the transmitter:

```
copy /proto/eccell /proto/icell
setfield /proto/icell transmitter GABA xtree_fg_req orange
```

Where, you might ask, are the cell icons that the **xtree** widget is supposed to display? The widget needs to be explicitly told to go and update its contents, using the *call* command to invoke its RESET action:

```
call /edit/lib/tree RESET
```

Now we have a couple of cells in the library. They get assigned a nice boring rectangular icon by default; this is the *shape* child element you would have seen if you did an “`le /edit/lib/tree`”. We can make them appear a little more interesting by creating an **xshape** as an icon for them:

```
create xshape /edit/lib/tree/shape -autoindex \
    -value compartment -pixflags v -pixflags c \
    -coords [-0.5,0,0] [0,0.5,0] [0.5,0,0] -drawmode FillPoly
call /edit/lib/tree RESET
```

Well, I only said a *little* more interesting. If you feel creative, you can put any fanciful shape you like to act as an icon for the cells. Note that the **xtree** widget uses the *value* field of the **xshapes** to decide which object classes the **xshapes** are meant to represent.

22.6.3 The Work Window

The next step is to set up the work window. In this case, the tree uses the *geometry* treemode, which displays elements according to their position in space. There are two function calls handled by this tree element. First, it calls a function to edit the cells when they are double-clicked (*edit_cell*). Second, it handles repositioning of cells in space by click-and-drag operations (*move_cell*).

```
create xcoredraw /edit/work [0,5:lib,100%,65%] \
    -xmin -4 -xmax 4 -ymin -4 -ymax 4
create xtree /edit/work/tree -path "/net/#/[TYPE=compartment]" \
    -treemode geometry \
    -script "edit_cell.D <v>; move_cell.w <d> <S> <x> <y>" 
copy /edit/lib/tree/shape[1] /edit/work/tree
```

We are almost ready to do the first of our *Netkit* operations: dragging cells from the library into the work window. We just need to write the *create_cell* function which the `/edit/lib/tree` widget will use:

```
create neutral /net // The new cells are created on top of /net
function create_cell(dest,srcval,x,y)
    str dest,srcval
    float x,y
    // Make sure that the destination is the work window
    if ({strcmp {dest} "/edit/work"} == 0)
```

```

copy {srcval} /net -autoindex
position ^ {x} {y} I
call /edit/work/# RESET
end
end

```

Note the use of the wildcard symbol “#” so that the RESET action will be called for all elements directly below */edit/work*. This is all you need in order to drag the prototype cells from the library into the work window. Go ahead and try it! If you list the child elements of */net*, (type “*le /net*”) you will see all the cells you have created, which of course correspond to the ones visible in the work window.

By now you should have dragged in enough cells to clutter up your work window, so it is time to write a little function that lets you reposition them. This function is very similar to the last one.

```

function move_cell(dest,srcval,x,y)
str dest,srcval
float x,y
// Make sure that the destination is the work window
if ({strcmp {dest} "/edit/work"} == 0)
    position {srcval} {x} {y} I
    call /edit/work/# RESET
end
end

```

22.6.4 Editing Cells

At this point, the cells you have created are just colored triangles in the work window. To make them a little more useful you have to be able to edit their fields. This requires three things: a parameter editor window, a function to update parameter values, and a function that calls up the parameter editor when a cell is double-clicked. In addition, there is a utility function *set_named_field* whose purpose is obvious. None of these functions is particularly exciting, so I present them without further comment.

```

function make_xedit_cell
create xform /parmedit/cell [500,500,500,150]
addfield /parmedit/cell elmpath -description "path of elm"
pushe /parmedit/cell
create xdialog name -title "Name"
create xdialog Em [0,0:name,50%,30] \
    -script "set_named_field <w> <v>"
create xdialog Vm [50%,0:name,50%,30] \
    -script "set_named_field <w> <v>" 
create xdialog inject -script "set_named_field <w> <v>" 

```

```

create xbutton UPDATE [0%,0:inject,50%,30] \
    -script "do_update_cellinfo"
create xbutton HIDE [50%,0:inject,50%,30] \
    -script "xhide /parmedit/cell"
pope
end
create neutral /parmedit // placeholder for param editors
make_xedit_cell // actually make the cell param editor

function set_named_field(widget,value)
    str widget,value
    str elm = {getfield {widget}/.. elmpath}
    str field = {getfield {widget} name}
    setfield {elm} {field} {value}
end

function do_update_cellinfo
    str cell = {getfield /parmedit/cell elmpath}
    setfield /parmedit/cell/name value {cell}
    setfield /parmedit/cell/Em value {getfield {cell} Em }
    setfield /parmedit/cell/Vm value {getfield {cell} Vm }
    setfield /parmedit/cell/inject value {getfield {cell} inject}
end

function edit_cell(reac)
    str reac
    setfield /parmedit/cell elmpath {reac}
    do_update_cellinfo
    xshowontop /parmedit/cell
end

```

Now you should be able to double-click on any of the cells and get a window with the vital statistics of the cell. If there is any interface-related lesson to be learned from this editor function, it is that the boilerplate code, which does really boring and basic stuff, takes up more space than all the interesting code put together.

22.6.5 Connecting Cells

To prove my last point, here we set up the function that does the far more interesting operation of connecting cells. First, the function itself:

```

function connect_cells(A,B)
    str A,B
    addmsg {A}/axon {B}/{getfield {A} transmitter} SPIKE

```

```
call /edit/work/# RESET
end
```

Then we set up the work tree to invoke the connect function. The arguments here include source and destination paths, valid message types and destination element types, a color for the arrows (which this particular tree does not actually have to deal with), a couple of flags, and then three script functions that are called in different situations. The syntax for the following example is explained in gory detail in the GENESIS Reference Manual.

```
call /edit/work/tree ADDMSGARROW "/net/##[TYPE=compartment]" \
"/net/##[TYPE=compartment]" all compartment green 0 0 \
"connect_cells.p <S> <D>" "" ""
```

There is a little complication here. In most situations, one would use a single tree to display elements, call functions to set up interconnections, and display the interconnections as arrows. If that were the case, we would be done by now. In this example, however, we want to interconnect cells, but the actual messages are between the axon and synapse child elements of the cells. So we create another **xtree** to display the messages as red arrows, and change its default *shape* icon to draw points so that it doesn't clutter up the screen.

```
create xtree /edit/work/arrows -path "/net/#[]/##[]" \
-treemode geometry \
-namemode "none" \
-pixflags c
call /edit/work/arrows ADDMSGARROW \
all all all all red 0 0 "" "" ""
setfield /edit/work/arrows/shape[0] drawmode DrawPoints
```

Now you can connect cells by the simple operation of clicking on cell A, and dragging it onto cell B. A red arrow appears between the two to represent the connection.

22.6.6 Plotting Cell Activity

By now you have a skeleton simulation builder, but it doesn't actually *do* anything! One possible way of watching what the simulation was doing would be to put an **xview** widget in the work window as well, and use colors to represent cellular activity. Here, though, we will draw graphs instead. All we need to do is set up a graph window, and write a function to generate a graph when a cell is dragged from the work window into the graph window. Drag in a few cells to see how it works.

```
create xform /graph [550,0,500,500]
create xgraph /graph/graph -hgeom 100% -ymin -0.1 -ymax 0.25 \
-xmin 0 -xmax 0.5 -yoffset 0.15 \
-script "add_plot.p <S>"
xshow /graph
```

```

function add_plot(src)
    str src
    str srcname = {getfield {src} name} @ " " \
        @ {getfield {src} index} @ ".Vm"
    create xplot /graph/graph/{srcname} -pixflags f
    addmsg {src} /graph/graph/{srcname} PLOT \
        Vm *{srcname} *{getfield {src} xtree_fg_req}
    useclock /graph/graph/{srcname} 2
end

```

22.6.7 Running Netkit

Almost everything is in place now. We still need to set up a few basic simulation parameters:

```

setclock 0 50e-6 // a 50 usec time step
setclock 1 1e-3
setclock 2 0.2e-3

```

If you want a quick preview, you can do the following:

```

reset
step 0.5 -time

```

To round off the network builder, let us put in a control panel as a final amenity. As with the cell parameter editor, this is also boring old boilerplate code.

```

function update_time
    setfield /control/currtime value {getstat -time}
end
create xform /control [0,0,500,100]
create xbutton /control/start [0,0,33%,30] \
    -script "step 0.5 -time"
create xbutton /control/reset [0:last,0,34%,30] \
    -script reset
create xbutton /control/stop [0:last,0,33%,30] -script stop
create xdialog /control/currtime \
    -label "Current time (sec)" -value 0
addaction /control/currtime PROCESS update_time
useclock /control/# 1
create xbutton /control/quit -script quit
xshow /control

```

And that is it for our network builder example. If you have correctly followed the steps outlined above, you should be rewarded with a display similar to that shown in Fig. 22.1. As promised, you now have a simulation-builder interface that manages a set of prototypes, uses click-and-drag operations for creating cells, connecting them and graphing them, and gives you access to your cell parameters.

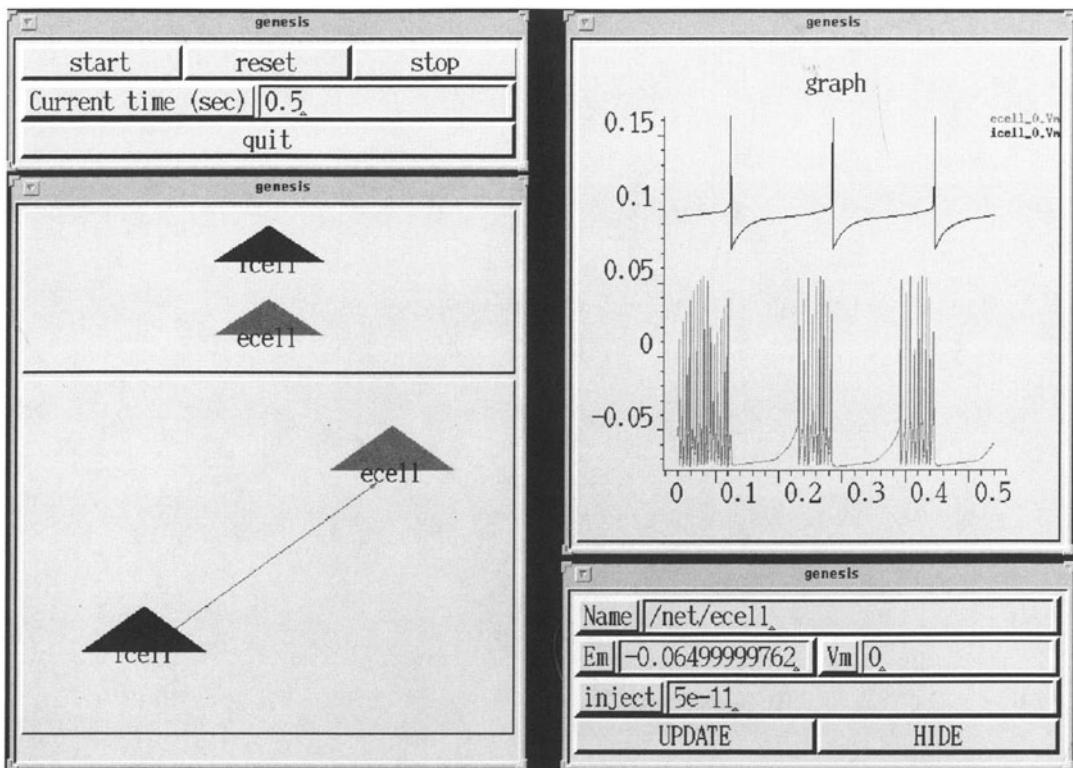


Figure 22.1 The display produced by the completed *Netkit* example. The general simulation controls are located in the upper left-hand corner. The edit window, containing the draw widgets for the library and work areas, is below them. The library widget contains icons for the excitatory (ecell) and inhibitory (icell) cell prototypes. The work widget contains the actual network. The graph widget is displayed on the upper right, and the parameter editing window (currently displaying the excitatory cell in the network) is below this. The simple network illustrated here has an inhibitory cell firing at its basal rate, connected to an excitatory cell that is being driven by 50 pA of injected current.

22.6.8 Extending Netkit

The version of *Netkit* you have just written is a very limited one, although it does a remarkable amount for under 150 lines of code. Although the tutorial format of the example blurs the organization somewhat, it should be apparent that the interface can be rather neatly divided into the general interface modules (like the edit and graph windows) and the component modules that handle the cell-specific operations of prototyping, interconnection and parameter editing. If you wanted to add an entirely new module, say, a module for delivering a repetitive stimulus, you would simply have to add a file with the functions for these three operations. This organization is clearly visible in the *Netkit* example files in the GENESIS distribution. The structure of the code should make it fairly clear how it could be extended:

1. There are obvious enhancements to the overall interface, such as a save and restore option, more control over simulation parameters such as clocks and the run time, and so on.
2. There are a whole slew of other network components that are not represented, for example, stimulus objects, globally applied neuromodulators, and so on. Each of these could be built up in the same sort of framework used for the single cell module.
3. The editing interface for the cell is presently very limited. There are a lot more parameters one might wish to change, such as the channel kinetics, synaptic weights, etc.
4. This leads towards a fundamental question: how should this network builder interface be linked into other related interfaces? For instance, for making and editing the cell prototypes one would want a complete neuron-building interface such as *Neurokit*, and for hooking together populations of cells perhaps we need yet another level of network interface.

I leave this example with these little “exercises for the reader.” The objective is not so much to lure you into an endless exercise in improving *Netkit*, which will probably be carried out by dedicated hackers long before you read this chapter. It is rather to get you thinking in terms of interfaces as modular constructs, and to see how the building blocks of XODUS can themselves be lumped into larger prefabricated modules that slot together to make pretty powerful interfaces.

22.7 Interface vs. Simulation

The most important part of an XODUS interface is the simulation. We have already stressed modularity in building an interface. The first step towards this is keeping the simulation and the interface separate. This applies both to the GENESIS code (scripts) that sets up the simulation, and to the hierarchy of the simulation itself. In other words, write separate functions for building the graphics and the simulation, and try to keep your simulation components on different element hierarchies from your interface components. This is important both from the viewpoint of managing the code, and also for efficiency. Graphics add overhead, no matter how efficiently you do them. In fact, graphics can easily consume more computer resources than the simulation itself. There are a few key points to keep in mind to minimize this overhead:

1. Although it may seem strange advice in a chapter on interfaces, one of the first things you are likely to want to do once you are in “production mode” with a simulation is to

turn off the interface. This is a good reason for keeping the interface and simulation separate!

2. Use slow clock rates for the graphical components. You rarely need to keep tabs on values more often than once in a hundred or so numerical time steps.
3. Use messages for passing data to widgets if they are being updated continuously. The draw, plot, dialog, var, view and cell widgets can all use messages as well as script value assignments. Messages are the native construct for shunting information around in GENESIS and are much faster than script-based updates. Messages are also readily parallelized in GENESIS, which may be important when you decide to run your simulation on multiple machines.

22.8 Summary

In this chapter you have met a menagerie of widgets, with a few examples for the simpler ones. We have seen how mouse and keyboard operations relate to actions and functions in the simulator. We then worked through the *Netkit* example where we saw several of the widgets in action, and learned about some basic design principles for XODUS-based interfaces. Beyond all this, the chapter has also had a hidden agenda, which may have been obvious to you. It is to persuade you to take the leap from learning about simulations, and trying out other people's models, to building your own. Having come this far, you have been exposed to a spectrum of neural simulation lore, ranging from theory and basic physiology to the specifics of GENESIS simulation objects. These are all just tools. Now go ahead and use them.

Appendix A

Acquiring and Installing GENESIS

A.1 System Requirements

GENESIS and its graphical front-end XODUS are written in C and run on SUN (SunOS 4 or Solaris 2), DECstation (Ulrix), Silicon Graphics (Irix 4.0.1 and up) or x86 PC (Linux or FreeBSD) machines with X-windows (versions X11R4, X11R5, and X11R6). IBM RS6000s (AIX), HPs (HPUX), DEC Alphas (OSF v2 and v3) and the Cray T3D have been successful in compiling and running, although our experience is limited. We welcome feedback on experiences with these platforms. Other platforms may be capable of running GENESIS, but the software has not been tested by Caltech outside of these environments. Although GENESIS may be ported to the Windows95 and NT operating systems at some time in the future, we currently recommend using the freely available Linux operating system for running GENESIS on a PC.

A.2 Using the CD-ROM

The CD-ROM included with this book contains the complete GENESIS version 2.1 distribution, which includes full source code and documentation for both GENESIS and XODUS, as well as the tutorial simulations described in this book. The CD-ROM also contains parallel GENESIS (PGENESIS) and a number of additional packages related to GENESIS that are usually available separately or via the World Wide Web. In addition to containing the GENESIS source code, the CD-ROM has precompiled binaries for the most common UNIX platforms. You may run these binaries directly from the CD-ROM. This will be useful if you are interested in trying the tutorials described in the book, or in evaluating the GENESIS

simulator. For regular use and best performance, the binaries can be installed on a hard disk.

Complete instructions for the use of the CD-ROM and the installation and running of GENESIS are given in the file *A-ReadMe.txt*. This information is also available in the hypertext file *A-ReadMe.html*, which may be viewed with a web browser, and which provides a convenient link to the hypertext GENESIS Reference Manual and other useful information.

A.3 Obtaining GENESIS over the Internet

GENESIS is continually evolving, and there will undoubtedly be new features incorporated into future versions. To be sure that you have the latest GENESIS distribution, and to learn about new developments, please check the GENESIS World Wide Web site (<http://www.bbb.caltech.edu/GENESIS>) or anonymous ftp site (*genesis.bbb.caltech.edu*). You may use these sites to download the latest versions of the software and documentation at no cost.

When using ftp to connect to *genesis.bbb.caltech.edu*, log in as the user “anonymous” and give your full email address as the password. You can then type “cd /pub/genesis” and download the software. Your first step should be to download the files *README* and *LATEST.NEWS*, with the commands “get README” and “get LATEST.NEWS”. These files will give further information about the current GENESIS version, and alert you to any new developments since the publication of this book. The *README* file will give further information on downloading and installing the files that are available. Typically, you will give the command “binary”, followed by the command “get *genesis.tar.Z*”. The file may take a while to transfer if you do this at a time when networks are busy. Finally, give the “quit” command.

The files mentioned above are directly accessible via hypertext links at the GENESIS web site. Information will also be available concerning “mirror” sites outside of the United States.

A.4 Installation and Documentation

GENESIS may be easily installed from the CD-ROM by using the installation script and instructions that are provided. To install GENESIS from a distribution that was obtained over the Internet, or to compile and install GENESIS on a platform for which there are no suitable precompiled binaries, you or your system administrator should change to the directory in which you wish the GENESIS directory tree to reside, and copy *genesis.tar.Z* to this directory. */usr* or */usr/local* would be a good location for this directory, although you may use your home directory or any other directory.

Then, give the UNIX command “`zcat genesis.tar.Z | tar xvf -`”. This will create the directory *genesis* and a number of subdirectories. Begin by reading the *README* file in the *genesis* directory. Directions for compiling and installing the software may be found in the *README* file contained in the *src* subdirectory. Directions for printing the GENESIS Reference Manual and installing the hypertext documentation may be found in *Doc/README*. The *Scripts/README* file describes the demonstration and tutorial simulations that are included with this distribution. Further inquiries concerning GENESIS or its installation may be addressed to *genesis@bbb.caltech.edu* by email.

Individuals or research groups who are considering using GENESIS as a research tool are strongly encouraged to join the GENESIS Users Group, BABEL. Information regarding BABEL membership may be obtained by email from *babel@bbb.caltech.edu*.

A.5 Copyright Notice

Copyright (C) 1997 by California Institute of Technology (Caltech)

Permission to use, copy, modify, and distribute this software and the included documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Caltech not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. Caltech makes no representations about the suitability or merchantability of this software for any purpose. It is provided “as is” without express or implied warranty.

Some components are copyrighted by the originating institution and are used with the permission of the authors. The conditions of these copyrights (none of which restrict the free distribution of GENESIS) appear with these modules.

Appendix B

GENESIS Script Listings

B.1 tutorial2.g

```

//genesis - tutorial2.g - GENESIS Version 2.0
/*=====
 A sample script to create a soma-like compartment. SI units are used.
=====*/
float PI = 3.14159

// soma parameters - chosen to be the same as in SQUID (but in SI units)
float RM = 0.33333      // specific membrane resistance (ohms m^2)
float CM = 0.01          // specific membrane capacitance (farads/m^2)
float RA = 0.3           // specific axial resistance (ohms m)
float EREST_ACT = -0.07  // resting membrane potential (volts)
float Eleak = EREST_ACT + 0.0106 // membrane leakage potential (volts)
float ENA   = 0.045       // sodium equilibrium potential
float EK    = -0.082       // potassium equilibrium potential

// cell dimensions (meters)
float soma_l = 30e-6     // cylinder equivalent to 30 micron sphere
float soma_d = 30e-6

float dt = 0.00005        // simulation time step in sec
setclock 0 {dt}           // set the simulation clock

//=====
// Function Definitions
//=====

```

```

function makecompartment(path, length, dia, Erest)
    str path
    float length, dia, Erest
    float area = length*PI*dia
    float xarea = PI*dia*dia/4

    create      compartment      {path}
    setfield   {path}          \
        Em      { Erest }   \           // volts
        Rm      { RM/area } \         // Ohms
        Cm      { CM*area } \         // Farads
        Ra      { RA*length/xarea } // Ohms
end

function make_Vmgraph
    float vmin = -0.100
    float vmax = 0.05
    float tmax = 0.100 // default simulation time = 100 msec
    create xform /data
    create xgraph /data/voltage
    setfield ^ xmax {tmax} ymin {vmin} ymax {vmax}
    create xbutton /data/RESET -script reset
    create xbutton /data/RUN  -script "step "{tmax}" -time"
    create xbutton /data/QUIT -script quit
    xshow /data
end

//=====
//      Main Script
//=====

create neutral /cell
// create the soma compartment "/cell/soma"
makecompartment /cell/soma {soma_l} {soma_d} {Eleak}

// provide current injection to the soma
setfield /cell/soma inject 0.3e-9 // 0.3 nA injection current

// make the graph to display soma Vm and pass messages to the graph
make_Vmgraph
addmsg /cell/soma /data/voltage PLOT Vm *volts *red

check
reset

```

B.2 tutorial3.g

```
//genesis - tutorial3.g - GENESIS Version 2.0
/*=====
A sample script to create a compartment containing channels taken from
hhchan.g in the neurokit prototypes library. SI units are used.
=====*/
include hhchan // functions to create Hodgkin-Huxley channels
/* hhchan.g assigns values to the global variables EREST_ACT, ENA, EK,
and SOMA_A. These will be superseded by values defined below. */
float PI = 3.14159

// soma parameters - chosen to be the same as in SQUID (but in SI units)
float RM = 0.33333           // specific membrane resistance (ohms m^2)
float CM = 0.01               // specific membrane capacitance (farads/m^2)
float RA = 0.3                // specific axial resistance (ohms m)
float EREST_ACT = -0.07        // resting membrane potential (volts)
float Eleak = EREST_ACT + 0.0106 // membrane leakage potential (volts)
float ENA   = 0.045            // sodium equilibrium potential
float EK    = -0.082            // potassium equilibrium potential

// cell dimensions (meters)
float soma_l = 30e-6          // cylinder equivalent to 30 micron sphere
float soma_d = 30e-6
float SOMA_A = soma_l*PI*soma_d // variable used by hhchan.g for soma area

float tmax = 0.1               // simulation time in sec
float dt = 0.00005             // simulation time step in sec
setclock 0 {dt}                // set the simulation clock

//=====
//      Function Definitions
//=====

function makecompartment(path, length, dia, Erest)
  str path
  float length, dia, Erest
  float area = length*PI*dia
  float xarea = PI*dia*dia/4

  create compartment {path}
  setfield {path} \
    Em    { Erest } \           // volts
    Rm    { RM/area } \         // Ohms
    Cm    { CM*area } \         // Farads
    Ra    { RA*length/xarea }  // Ohms
end
```

```

function step_tmax
    step {tmax} -time
end

//=====
//    Graphics Functions
//=====

function make_control
    create xform /control [10,50,250,145]
    create xlabel /control/label -hgeom 50 -bg cyan -label "CONTROL PANEL"
    create xbutton /control/RESET -wgeom 33%      -script reset
    create xbutton /control/RUN   -xgeom 0:RESET -ygeom 0:label -wgeom 33% \
        -script step_tmax
    create xbutton /control/QUIT -xgeom 0:RUN   -ygeom 0:label -wgeom 34% \
        -script quit
    create xdialog /control/Injection -label "Injection (amperes)" \
        -value 0.3e-9 -script "set_inject <widget>"
    xshow /control
end

function make_Vmgraph
    float vmin = -0.100
    float vmax = 0.05
    create xform /data [265,50,350,350]
    create xlabel /data/label -hgeom 10% -label "Soma with Na and K Channels"
    create xgraph /data/voltage -hgeom 90% -title "Membrane Potential"
    setfield ^ XUnits sec YUnits Volts
    setfield ^ xmax {tmax} ymin {vmin} ymax {vmax}
    xshow /data
end

function set_inject(dialog)
    str dialog
    setfield /cell/soma inject {getfield {dialog} value}
end

//=====
//    Main Script
//=====

create neutral /cell
// create the soma compartment "/cell/soma"
makecompartment /cell/soma {soma_1} {soma_d} {Eleak}
setfield /cell/soma initVm {EREST_ACT} // initialize Vm to rest potential

// provide current injection to the soma
setfield /cell/soma inject 0.3e-9      // 0.3 nA injection current

```

```

// Create two channels, "/cell/soma/Na_squid_hh" and "/cell/soma/K_squid_hh"
pushe /cell/soma
make_Na_squid_hh
make_K_squid_hh
pope

// The soma needs to know the value of the channel conductance
// and equilibrium potential in order to calculate the current
// through the channel. The channel calculates its conductance
// using the current value of the soma membrane potential.

addmsg /cell/soma/Na_squid_hh /cell/soma CHANNEL Gk Ek
addmsg /cell/soma /cell/soma/Na_squid_hh VOLTAGE Vm
addmsg /cell/soma/K_squid_hh /cell/soma CHANNEL Gk Ek
addmsg /cell/soma /cell/soma/K_squid_hh VOLTAGE Vm

// make the control panel
make_control

// make the graph to display soma Vm and pass messages to the graph
make_Vmgraph
addmsg /cell/soma /data/voltage PLOT Vm *volts *red

check
reset

```

B.3 tutorial4.g

```

//genesis - tutorial4.g - GENESIS Version 2.0

/*=====
A sample script to create a multicompartmental neuron with synaptic
input. SI units are used.
=====*/
include hhchan          // functions to create Hodgkin-Huxley channels
/* hhchan.g assigns values to the global variables EREST_ACT, ENA, EK,
   and SOMA_A. These will be superseded by values defined below. */
float PI = 3.14159

// soma parameters - chosen to be the same as in SQUID (but in SI units)
float RM = 0.33333      // specific membrane resistance (ohms m^2)
float CM = 0.01          // specific membrane capacitance (farads/m^2)
float RA = 0.3            // specific axial resistance (ohms m)
float EREST_ACT = -0.07    // resting membrane potential (volts)
float Eleak = EREST_ACT + 0.0106 // membrane leakage potential (volts)

```

```

float ENA    = 0.045          // sodium equilibrium potential
float EK     = -0.082         // potassium equilibrium potential

// cell dimensions (meters)
float soma_l = 30e-6          // cylinder equivalent to 30 micron sphere
float soma_d = 30e-6
float dend_l = 100e-6          // we will add a 100 micron long dendrite
float dend_d = 2e-6            // give it a 2 micron diameter
float SOMA_A = soma_l*PI*soma_d // variable used by hhchan.g for soma area

float gmax = 5e-10             // maximum synaptic conductance (Siemen)

float tmax = 0.1               // simulation time in sec
float dt = 0.00005             // simulation time step in sec
setclock 0 {dt}                // set the simulation clock

//=====
//      Function Definitions
//=====

function makecompartment(path, length, dia, Erest)
  str path
  float length, dia, Erest
  float area = length*PI*dia
  float xarea = PI*dia*dia/4

  create compartment {path}
  setfield {path} \
    Em   { Erest } \           // volts
    Rm   { RM/area } \         // Ohms
    Cm   { CM*area } \         // Farads
    Ra   { RA*length/xarea } \ // Ohms
end

function makechannel(compartment,channel,Ek,tau1,tau2,gmax)
  str compartment
  str channel
  float Ek                      // Volts
  float tau1,tau2                // sec
  float gmax                     // Siemens (1/ohms)

  create synchan {compartment}/{channel}
  setfield ^ \
    Ek           {Ek} \
    tau1        {tau1} \
    tau2        {tau2} \
    gmax        {gmax}
  addmsg {compartment}/{channel} {compartment} CHANNEL Gk Ek
  addmsg {compartment} {compartment}/{channel} VOLTAGE Vm

```

```
end

function makeneuron(path, soma_l, soma_d, dend_l, dend_d)
    str path
    float soma_l, soma_d, dend_l, dend_d

    create neutral {path}
    makecompartment {path}/soma {soma_l} {soma_d} {ELeak}
    setfield /cell/soma initVm {EREST_ACT}

// Create two channels, "{path}/soma/Na_squid_hh" and "{path}/soma/K_squid_hh"
    pushes {path}/soma
    make_Na_squid_hh
    make_K_squid_hh
    pope

    // The soma needs to know the value of the channel conductance
    // and equilibrium potential in order to calculate the current
    // through the channel. The channel calculates its conductance
    // using the current value of the soma membrane potential.
    addmsg {path}/soma/Na_squid_hh {path}/soma CHANNEL GN EK
    addmsg {path}/soma {path}/soma/Na_squid_hh VOLTAGE Vm
    addmsg {path}/soma/K_squid_hh {path}/soma CHANNEL GK EK
    addmsg {path}/soma {path}/soma/K_squid_hh VOLTAGE Vm

    // make the dendrite compartment and link it to the soma
    makecompartment {path}/dend {dend_l} {dend_d} {EREST_ACT}
    makechannel {path}/dend Ex_channel {ENA} 0.003 0.003 {gmax}
    addmsg {path}/dend {path}/soma RAXIAL Ra previous_state
    addmsg {path}/soma {path}/dend AXIAL previous_state

// add a spike generator to the soma
    create spikegen {path}/soma/spike
    setfield {path}/soma/spike thresh 0 abs_refract 0.010 output_amp 1
    /* use the soma membrane potential to drive the spike generator */
    addmsg {path}/soma {path}/soma/spike INPUT Vm
end // makeneuron

function step_tmax
    step {tmax} -time
end

function makeinput(path)
    str path
    int msgnum
    create randomspike /randomspike
    setfield ^ min_amp 1.0 max_amp 1.0 rate 200 reset 1 reset_value 0
    addmsg /randomspike {path} SPIKE
    msgnum = {getfield {path} nsynapses} - 1
```

```

setfield {path} \
    synapse[{\msgnum}].weight 1 synapse[{\msgnum}].delay 0
addmsg /randomspike /data/voltage \
    PLOTSCALE state *input *blue 0.01 0
//                                title color scale offset
end

//=====
//    Graphics Functions
//=====

function make_control
    create xform /control [10,50,250,145]
    create xlabel /control/label -hgeom 50 -bg cyan -label "CONTROL PANEL"
    create xbutton /control/RESET -wgeom 33%      -script reset
    create xbutton /control/RUN   -xgeom 0:RESET -ygeom 0:label -wgeom 33% \
        -script step_tmax
    create xbutton /control/QUIT -xgeom 0:RUN -ygeom 0:label -wgeom 34% \
        -script quit
    create xdialog /control/Injection -label "Injection (amperes)" \
        -value 0.0 -script "set_inject <widget>"
    create xtoggle /control/feedback -script toggle_feedback
    setfield /control/feedback offlabel "Feedback OFF" \
        onlabel "Feedback ON" state 1
    xshow /control
end

function make_Vmgraph
    float vmin = -0.100
    float vmax = 0.05
    create xform /data [265,50,350,350]
    create xlabel /data/label -hgeom 10% -label "Simple Neuron Model"
    create xgraph /data/voltage -hgeom 90% -title "Membrane Potential"
    setfield ^ XUnits sec YUnits Volts
    setfield ^ xmax {tmax} ymin {vmin} ymax {vmax}
    xshow /data
end

function set_inject(dialog)
    str dialog
    setfield /cell/soma inject {getfield {dialog} value}
end

function make_condgraph
    create xform /condgraphs [620,50,475,350]
    pushe /condgraphs
    create xgraph channel_Gk -hgeom 100% -title "Channel Conductance"
    setfield channel_Gk xmin 0 xmax {tmax} ymin 0 ymax {gmax*10}
    setfield channel_Gk XUnits "sec" YUnits "Gk (Siemen)"

```

```
pope
xshow /condgraphs
end

function toggle_feedback
    int msgnum
    if ({getfield /control/feedback state} == 0)
        deletemsg /cell/soma/spike 0 -out
        echo "Feedback connection deleted"
    else
        addmsg /cell/soma/spike /cell/dend/Ex_channel SPIKE
        msgnum = {getfield /cell/dend/Ex_channel nsynapses} - 1
        setfield /cell/dend/Ex_channel \
            synapse[{msgnum}].weight 10 synapse[{msgnum}].delay 0.005
        echo "Feedback connection added"
    end
end

//=====
//      Main Script
//=====

// create the neuron "/cell"
makeneuron /cell {soma_1} {soma_d} {dend_1} {dend_d}
setfield /cell/soma inject 0.0

// make the control panel
make_control

// make the graph to display soma Vm and pass messages to the graph
make_Vmgraph
addmsg /cell/soma /data/voltage PLOT Vm *volts *red

makeinput /cell/dend/Ex_channel

make_condgraph
addmsg /cell/dend/Ex_channel /condgraphs/channel_Gk PLOT Gk *Gk *black

// finally, we add some feedback from the axon to the dendrite
addmsg /cell/soma/spike /cell/dend/Ex_channel SPIKE
setfield /cell/dend/Ex_channel \
    synapse[1].weight 10 synapse[1].delay 0.005

check
reset
```

B.4 tutorial5.g

```
//genesis - tutorial5.g - GENESIS Version 2.0

/*=====
A sample script which uses the cell reader to create a
multicompartmental neuron with synaptic input. SI units are used.
=====*/

// Create a library of prototype elements to be used by the cell reader
include protodefs

float gmax = 5e-10           // maximum synaptic conductance (Siemen)

float tmax = 0.1              // simulation time in sec
float dt = 0.00005            // simulation time step in sec
setclock 0 {dt}               // set the simulation clock

//=====
//      Function Definitions
//=====

function step_tmax
    step {tmax} -time
end

function makeinput(path)
    str path
    int msgnum
    create randomspike /randomspike
    setfield ^ min_amp 1.0 max_amp 1.0 rate 200 reset 1 reset_value 0
    addmsg /randomspike {path} SPIKE
    msgnum = {getfield {path} nsynapses} - 1
    setfield {path} \
        synapse[{msgnum}].weight 1 synapse[{msgnum}].delay 0
    addmsg /randomspike /data/voltage \
        PLOTSCALE state *input *blue 0.01 0
    //                      title color scale offset
end

//=====
//      Graphics Functions
//=====

function make_control
    create xform /control [10,50,250,145]
    create xlabel /control/label -hgeom 50 -bg cyan -label "CONTROL PANEL"
    create xbutton /control/RESET -wgeom 33%      -script reset
    create xbutton /control/RUN   -xgeom 0:RESET -ygeom 0:label -wgeom 33% \
```

```
-script step_tmax
create xbutton /control/QUIT -xgeom 0:RUN -ygeom 0:label -wgeom 34% \
    -script quit
create xdialog /control/Injection -label "Injection (amperes)" \
    -value 0.0 -script "set_inject <widget>" 
create xtoggle /control/feedback -script toggle_feedback
setfield /control/feedback offlabel "Feedback OFF" \
    onlabel "Feedback ON" state 1
xshow /control
end

function make_Vmgraph
    float vmin = -0.100
    float vmax = 0.05
    create xform /data [265,50,350,350]
    create xlabel /data/label -hgeom 10% -label "Simple Neuron Model"
    create xgraph /data/voltage -hgeom 90% -title "Membrane Potential"
    setfield ^ XUnits sec YUnits Volts
    setfield ^ xmax {tmax} ymin {vmin} ymax {vmax}
    xshow /data
end

function set_inject(dialog)
    str dialog
    setfield /cell/soma inject {getfield {dialog} value}
end

function make_condgraph
    create xform /condgraphs [620,50,475,350]
    pushe /condgraphs
    create xgraph channel_Gk -hgeom 100% -title "Channel Conductance"
    setfield channel_Gk xmin 0 xmax {tmax} ymin 0 ymax {gmax*10}
    setfield channel_Gk XUnits "sec" YUnits "Gk (siemens)"
    pope
    xshow /condgraphs
end

function toggle_feedback
    int msgnum
    if ({getfield /control/feedback state} == 0)
        deletemsg /cell/soma/spike 0 -out
        echo "Feedback connection deleted"
    else
        addmsg /cell/soma/spike /cell/dend/Ex_channel SPIKE
        msgnum = {getfield /cell/dend/Ex_channel nsynapses} - 1
        setfield /cell/dend/Ex_channel \
            synapse[{msgnum}].weight 10 synapse[{msgnum}].delay 0.005
        echo "Feedback connection added"
    end
end
```

```

end

//=====
//      Main Script
//=====

// Build the cell from a parameter file using the cell reader
readcell cell.p /cell

setfield /cell/soma inject 0.0

// make the control panel
make_control

// make the graph to display soma Vm and pass messages to the graph
make_Vmgraph
addmsg /cell/soma /data/voltage PLOT Vm *volts *red

makeinput /cell/dend/Ex_channel           // Create synaptic inputs

// Make synaptic conductance graph and pass message to the graph
make_condgraph
addmsg /cell/dend/Ex_channel /condgraphs/channel_Gk PLOT Gk *Gk *black

// finally, we add some feedback from the axon to the dendrite
addmsg /cell/soma/spike /cell/dend/Ex_channel SPIKE
setfield /cell/dend/Ex_channel \
    synapse[1].weight 10 synapse[1].delay 0.005

check
reset

```

B.5 hhchan.g

```

// genesis 2.0

/* FILE INFORMATION
** hh_channel implementation of squid giant axon voltage-dependent
** channels, according to :
** A.L.Hodgkin and A.F.Huxley, J.Physiol(Lond) 117, pp 500-544 (1952)
**
** This file depends on functions and constants defined in library.g
*/

// CONSTANTS
float EREST_ACT = -0.060 /* granule cell resting potl */
float ENA = 0.045
float EK = -0.090
float SOMA_A = 1e-9          /* Square meters */

```

```

int EXPONENTIAL =    1
int SIGMOID      =    2
int LINOID        =    3

//*****************************************************************************
Some conventions in using the HH_CHANNELS

HH_CONVENTIONS
=====
Activation state variable is called x for all channels
Inactivation state variable is called y for all channels
In the traditional hh notations: x=m, y=h for Na channel; x=n for K_channel

There are three functional forms for alpha and beta for each state variable:
FORM 1: alpha(v) = A exp((v-V0)/B)                                (EXPONENTIAL)
FORM 2: alpha(v) = A / (exp((v-V0)/B) + 1)                          (SIGMOID)
FORM 3: alpha(v) = A (v-V0) / (exp((v-V0)/B) - 1)                  (LINOID)
The same functional forms are used for beta.
In the simulator, the FORM, A, B and V0 are designated by:
X_alpha_FORM, X_alpha_A, X_alpha_B, X_alpha_V0   alpha function for state var x
X_beta_FORM, X_beta_A, X_beta_B, X_beta_V0     beta function for state var x
Y_alpha_FORM, Y_alpha_A, Y_alpha_B, Y_alpha_V0   alpha function for state var y
Y_beta_FORM, Y_beta_A, Y_beta_B, Y_beta_V0     beta function for state var y

The conductance is calculated as g = Gbar*x^Xpower * y^Ypower
For a squid axon Na channel: Xpower = 3, Ypower = 1 (m^3 h)
                           K channel: Xpower = 4, Ypower = 0 (n^4)

These are linked to the soma by two messages :
addmsg /soma/hh_channel /soma CHANNEL Gk Ek
addmsg /soma /soma/hh_channel VOLTAGE Vm

//*****************************************************************************
// Original Hodgkin-Huxley squid parameters, implemented as hh_channel elements

//=====
//          ACTIVE SQUID NA CHANNEL
//          A.L.Hodgkin and A.F.Huxley, J.Physiol(Lond) 117, pp 500-544 (1952)
//=====

function make_Na_squid_hh
    if ({exists Na_squid_hh})
        return
    end

    create          hh_channel       Na_squid_hh
    setfield Na_squid_hh \
        Ek           {ENA} \           // V

```

```

Gbar           { 1.2e3 * SOMA_A } \
Xpower        3.0 \
Ypower        1.0 \
X_alpha_FORM {LINOID} \
X_alpha_A    -0.1e6 \
X_alpha_B    -0.010 \
X_alpha_V0   { 0.025 + EREST_ACT } \
X_beta_FORM  {EXPONENTIAL} \
X_beta_A     4.0e3 \
X_beta_B     -18.0e-3 \
X_beta_V0   { 0.0 + EREST_ACT } \
Y_alpha_FORM {EXPONENTIAL} \
Y_alpha_A    70.0 \
Y_alpha_B    -20.0e-3 \
Y_alpha_V0   { 0.0 + EREST_ACT } \
Y_beta_FORM  {SIGMOID} \
Y_beta_A     1.0e3 \
Y_beta_B     -10.0e-3 \
Y_beta_V0   { 30.0e-3 + EREST_ACT } \
end

//=====
//          ACTIVE K CHANNEL - SQUID
//          A.L.Hodgkin and A.F.Huxley, J.Physiol(Lond) 117, pp 500-544 (1952)
//=====

function make_K_squid_hh
  if ({exists K_squid_hh})
    return
  end

  create      hh_channel      K_squid_hh
  setfield K_squid_hh \
    Ek          {EK} \
    Gbar        {360.0*SOMA_A} \
    Xpower      4.0 \
    Ypower      0.0 \
    X_alpha_FORM {LINOID} \
    X_alpha_A   -10.0e3 \
    X_alpha_B   -10.0e-3 \
    X_alpha_V0  {10.0e-3+EREST_ACT} \
    X_beta_FORM {EXPONENTIAL} \
    X_beta_A    125.0 \
    X_beta_B    -80.0e-3 \
    X_beta_V0   {0.0+EREST_ACT}
end

```

B.6 hhchan_K.g

```
//genesis - hhchan_K.g - creates an extended object for a H-H K channel

// Define some constants to define the form of the rate constant equation
int EXPONENTIAL = 1
int SIGMOID = 2
int LINOID = 3

// Original Hodgkin-Huxley squid parameters
float EREST_ACT = -0.070
float EK = -0.082

create hh_channel K_squid_hh
setfield K_squid_hh \
    Ek {EK} \ // V
    Gbar 0.0 \ 
    Xpower 4.0 \
    Ypower 0.0 \
    X_alpha_FORM {LINOID} \
    X_alpha_A -10.0e3 \ // 1/V-sec
    X_alpha_B -10.0e-3 \ // V
    X_alpha_V0 {10.0e-3+EREST_ACT} \ // V
    X_beta_FORM {EXPONENTIAL} \
    X_beta_A 125.0 \ // 1/sec
    X_beta_B -80.0e-3 \ // V
    X_beta_V0 {0.0+EREST_ACT} // V

addfield K_squid_hh gdens // conductance density
setfield K_squid_hh gdens 360.0 // S/m^2

setfieldprot K_squid_hh -hidden Xpower Ypower X_alpha_FORM \
    X_alpha_A X_alpha_B X_alpha_V0 X_beta_FORM X_beta_A \
    X_beta_B X_beta_V0 Y_alpha_FORM Y_alpha_A Y_alpha_B \
    Y_alpha_V0 Y_beta_FORM Y_beta_A Y_beta_B Y_beta_V0
setfieldprot K_squid_hh -readonly Gbar

function K_squid_hh_SET(action, field, newvalue)
    float newvalue
    float PI = 3.14159
    if (field == "gdens")
        setfield . Gbar {PI*{getfield .. dia}*{getfield .. len}*newvalue}
    end
    return 0 // indicate that SET action isn't yet complete
end
addaction K_squid_hh SET K_squid_hh_SET

function K_squid_hh_CREATE(action, parent, object, elm)
    float PI = 3.14159
```

```

if (!{isa compartment ..})
    echo K_squid_hh must be the child of a compartment
    return 0
end
setfield . Gbar {PI*{getfield .. dia}*{getfield .. len}*{getfield . gdens}}
addmsg . . . CHANNEL Gk Ek
addmsg . . . VOLTAGE Vm
return 1
end
addaction K_squid_hh CREATE K_squid_hh_CREATE

addobject K_squid_hh K_squid_hh -author "J. R. Hacker" \
    -description "Hodgkin-Huxley Active K Squid Channel - SI units"

```

B.7 userprefs.g

```

// genesis

// Customized userprefs.g to run the "Tutorial 5" simulation with neurokit

/*****
**
**      DO NOT EDIT THIS FILE IN THE neurokit DIRECTORY!
**
**      Make a copy of this file in every directory that contains .p
**      files and edit the copies, in order to customize neurokit for
**      different simulations. When you run neurokit from other
**      directories, the simulator will look for the local version of
**      userprefs.g, and if it cannot find it there will look for the
**      default in the neurokit directory.
**
**      There are three aspects to customization :
**
**          1      Include the appropriate script files from the /neuron/prototype
**                  directory and from wherever you have defined new prototype
**                  elements.
**
**          2      Invoke the functions that make the prototypes you want for
**                  your simulation.
**
**          3      Put your preferences for the user_variables defined in
**                  defaults.g in the copies of this file.
**
*****/

echo Using local user preferences

/*****

```

```
**      1      Including script files for prototype functions
*****  
  
/* file for standard compartments */
include compartments  
  
/* file for Hodgkin-Huxley Squid Na and K channels */
include hhchan  
  
/* file for synaptic channels */
include synchans  
  
/* file which makes a spike generator */
include protospike  
  
*****  
**      2  Invoking functions to make prototypes in the /library element
*****  
  
/* To ensure that all subsequent elements are made in the library */
pushe /library  
  
    make_cylind_compartment          /* makes "compartment" */  
  
/* Assign some constants to override those used in hhchan.g */
EREST_ACT = -0.07           // resting membrane potential (volts)
ENA    = 0.045              // sodium equilibrium potential
EK     = -0.082             // potassium equilibrium potential  
  
make_Na_squid_hh      /* makes "Na_squid_hh" */
make_K_squid_hh      /* makes "K_squid_hh" */
make_Ex_channel       /* synchan with Ek = 0.045, tau1 = tau2 = 3 msec */
  
/* In case we need it later, put an inhibitory GABA-activated channel
   in the library, too */
  
make_Inh_channel     /* synchan: Ek = -0.082, tau1 = tau2 = 20 msec */
make_spike          /* Make a spike generator element */
pope /            /* return to the root element */  
  
*****  
**      3      Setting preferences for user-variables.
*****  
  
/* See defaults.g for default values of these */
```

```

str user_help = "../neurokit/README"

user_cell = "/cell"
user_pfile = "cell.p"

user_runtime = 0.1
user_dt = 50e-6 // 0.05 msec
user_refresh = 5

// These are used for the two buttons which can be used to enter a value
// in the "Syn Type" dialog box
user_syntype1 = "Ex_channel"
user_syntype2 = "Inh_channel"

user_inject = 0.3 // default injection current (nA)

// Set the scales for the graphs in the two cell windows
user_ymin1 = -0.1
user_ymax1 = 0.05
user_xmax1 = 0.1
user_xmax2 = 0.1
user_ymin2 = 0.0
user_ymax2 = 5e-9

/* This displays the second cell window and plots the "Gk" field of the
   "Inh_channel" channel for the compartment in which a recording electrode
   has been planted. The default values of the field and path
   are "Vm" and ".", meaning to plot the Vm field for the compartment
   which is selected for recording.
*/
user_numxouts = 2
user_field2      = "Gk"
user_path2 = "Ex_channel"

```

B.8 cellproto.g

```

// genesis 2.0 - prototype cell for the Netkit example
float EREST_ACT = -0.065

function make_cell
    create compartment /proto/cell
        setfield /proto/cell \
            Rm 3.2e9 \
            Cm 6.3e-12 \
            Ra 16e6 \
            Em -0.065

```

```
create tabchannel /proto/cell/Na // set up the Na channel
    setfield /proto/cell/Na \
        Ek 0.045      Gbar 8e-7   Xpower 3     Ypower 1     Zpower 0
    setupalpha /proto/cell/Na X \ // setting up the X gate
        {320e3 * (0.013 + EREST_ACT)} \
        -320e3 -1.0 {-1.0 * (0.013 + EREST_ACT)} -0.004 \
        {-280e3 * (0.040 + EREST_ACT)} \
        280e3 -1.0 {-1.0 * (0.040 + EREST_ACT)} 5.0e-3 \
        -size 1000 -range -0.1 0.05

    setupalpha /proto/cell/Na Y \ // setting up the Y gate
        128 0 0 {-1.0 * (0.017 + EREST_ACT)} 0.018 \
        4e3 0 1 {-1.0 * (0.040 + EREST_ACT)} -5e-3 \
        -size 1000 -range -0.1 0.05

create tabchannel /proto/cell/K // set up the K channel
    setfield /proto/cell/K \
        Ek -0.090      Gbar 2.4e-7   Xpower 4     Ypower 0     Zpower 0
    setupalpha /proto/cell/K X \ // setting up the X gate
        {32e3 * (0.015 + EREST_ACT)} \
        -32e3 -1 {-1.0 * (0.015 + EREST_ACT)} -5e-3, \
        500 0 0 {-1.0 * (0.010 + EREST_ACT)} 40e-3 \
        -size 1000 -range -0.1 0.05

addmsg /proto/cell /proto/cell/Na VOLTAGE Vm
addmsg /proto/cell/Na /proto/cell CHANNEL Gk Ek

addmsg /proto/cell /proto/cell/K VOLTAGE Vm
addmsg /proto/cell/K /proto/cell CHANNEL Gk Ek

create spikegen /proto/cell/axon
setfield /proto/cell/axon abs_refract 0.001 thresh 0.0 output_amp 1
addmsg /proto/cell /proto/cell/axon INPUT Vm

create synchan /proto/cell/glu
    setfield /proto/cell/glu \
        tau1 2e-3       tau2 2e-3 \
        gmax 1e-7       Ek 0.045
addmsg /proto/cell /proto/cell/glu VOLTAGE Vm
addmsg /proto/cell/glu /proto/cell CHANNEL Gk Ek

create synchan /proto/cell/GABA
    setfield /proto/cell/GABA \
        tau1 20e-3      tau2 20e-3 \
        gmax 2e-8       Ek -0.090
addmsg /proto/cell /proto/cell/GABA VOLTAGE Vm
addmsg /proto/cell/GABA /proto/cell CHANNEL Gk Ek

end
```

Bibliography

- Acton, F. S. (1970). *Numerical Methods That Work*, Harper and Row, New York.
- Adams, D. J. and Gage, P. W. (1979a). Characteristics of sodium and calcium conductance changes produced by membrane depolarization in an *Aplysia* neurone, *J. Physiol.* **289**: 143–161.
- Adams, D. J. and Gage, P. W. (1979b). Ionic currents in response to membrane depolarization in an *Aplysia* neurone, *J. Physiol.* **289**: 115–141.
- Adams, D. J., Smith, S. J. and Thompson, S. H. (1980). Ionic currents in molluscan soma, *Ann. Rev. Neurosci.* **3**: 141–167.
- Adams, P. R. (1992). The platonic neuron gets the hots, *Current Biology* **5**: 625–627.
- Adams, W. B. and Levitan, I. B. (1985). Voltage and ion dependences of the slow currents which mediate bursting in *Aplysia* neurone R15, *J. Physiol.* **360**: 69–93.
- Adrian, E. D. (1942). Olfactory reactions in the brain of the hedgehog, *J. Physiol. (London)* **100**: 459–473.
- Agmon-Snir, H. and Segev, I. (1993). Signal delay and input synchronization in passive dendritic structures, *J. Neurophysiol.* **70**: 2066–2085.
- Alberts, B., Bray, D., Lewis, J., Raff, M., Roberts, K. and Watson, J. (1994). *Molecular Biology of the Cell*, third edn., Garland Publishing Inc., New York (Especially Chapter 15).
- Aldrich, R. W., Getting, P. A. and Thompson, S. H. (1979a). Inactivation of delayed outward current in molluscan neurone somata, *J. Physiol.* **291**: 507–530.

- Aldrich, R. W., Getting, P. A. and Thompson, S. H. (1979b). Mechanism of frequency-dependent broadening of molluscan neurone soma spikes, *J. Physiol.* **291**: 531–544.
- Alevizos, A., Weiss, K. R. and Koester, J. (1991). Synaptic actions of identified peptidergic neuron R15 in *Aplysia*. I. Activation of respiratory pumping, *J. Neurosci.* **11**: 1263–1274.
- Alper, J. (1994). Scientists return to the elementary-school classroom, *Science* **264**: 768–769.
- Azencott, R. (ed.) (1992). *Simulated Annealing: Parallelization Techniques*, Wiley, New York.
- Barbour, B. (1993). Synaptic currents evoked in Purkinje cells by stimulating individual granule cells, *Neuron* **11**: 759–769.
- Barker, J. L. and Smith, T. G. (1978). Electrophysiological studies of molluscan neurons generating bursting pacemaker potential activity, in P. Chalozinitas and D. Boisson (eds), *Abnormal Neuronal Discharges*, Raven Press, N. Y., pp. 359–387.
- Belliveau, J., McKinstry, R., Buchbinder, B., Weisskoff, R., Cohen, M., Vevea, J., Brady, T. and Rosen, B. (1991). Functional mapping of the human visual cortex by magnetic resonance imaging, *Science* **254**: 716–719.
- Benson, J. A. and Adams, W. B. (1989). Ionic mechanisms of endogenous activity in molluscan burster neurons, in J. W. Jacklet (ed.), *Neuronal and Cellular Oscillators*, Marcel Dekker, Albany, NY, chapter 4, pp. 87–120.
- Bernander, O., Douglas, R. D., Martin, K. A. and Koch, C. (1991). Synaptic background activity influences spatiotemporal integration in single pyramidal cells, *Proc. Natl. Acad. Sci. (USA)* **88**: 11569–11573.
- Bernstein, J. (1902). Untersuchungen zur thermodynamik der bioelektrischen ströme. erster theil., *Pflügers Arch.* **82**: 521–562.
- Bertram, R. (1993). A computational study of the effects of serotonin on a molluscan burster neuron, *Biol. Cybernetics* **69**: 257–267.
- Bhalla, U. S. and Bower, J. M. (1993). Exploring parameter space in detailed single neuron models: Simulations of the mitral and granule cells of the olfactory bulb, *J. Neurophysiol.* **69**: 1948–1965.
- Bhalla, U. S. and Bower, J. M. (1997). Multi-day recordings from olfactory bulb neurons in awake freely moving rats: Spatial and temporally organized variability in odorant response properties, *J. Comp. Neurosci.* **4**: 221–256.

- Bhalla, U. S. and Iyengar, R. (1997). Models of biochemical signaling pathways in neurons. I. feedback loop involving PKC and the MAPK cascade. (To be submitted).
- Bhalla, U. S., Bilitch, D. H. and Bower, J. M. (1992). Rallpacks: A set of benchmarks for neuronal simulators, *Trends Neurosci.* **15**: 453–458.
- Biedenbach, M. A. (1966). Effects of anesthetics and cholinergic drugs on prepyriform electrical activity in cats, *Exp. Neurol.* **16**: 464–479.
- Biedenbach, M. A. and Stevens, C. F. (1969a). Electrical activity in cat olfactory cortex produced by synchronous orthodromic volleys, *J. Neurophysiol.* **32**: 193–203.
- Biedenbach, M. A. and Stevens, C. F. (1969b). Electrical activity in cat olfactory cortex as revealed by intracellular recording, *J. Neurophysiol.* **32**: 204–214.
- Bloomfield, S. A., Hamos, J. E. and Sherman, S. M. (1987). Passive cable properties and morphological correlates of neurones in the lateral geniculate nucleus of the cat, *J. Physiol. (London)* **383**: 653–692.
- Bower, J. M. (1992). Modeling the nervous system, *Trends Neurosci.* **15**: 411–412.
- Bower, J. M. (1995). Reverse engineering the nervous system: An *in vivo*, *in vitro*, and *in computo* approach to understanding the mammalian olfactory system, in S. F. Zornetzer, J. L. Davis and C. Lau (eds.), *An Introduction to Neural and Electronic Networks*, second edn., Academic Press, New York, NY, pp. 3–28.
- Bower, J. M. (1997a). Is the cerebellum sensory for motor's sake, or motor for sensory's sake: The view from the whiskers of a rat?, in C. de Zeeuw, P. Strata and J. Voogd (eds.), *Progress Brain Res.*, Vol. 114, pp. 483–516.
- Bower, J. M. (1997b). The cerebellum and the control of sensory data aquisition, in J. Schmahmann (ed.), *The Cerebellum and Cognition*, Academic Press, San Diego, pp. 489–513.
- Bray, D. (1995). Protein molecules as computational elements in living cells, *Nature* **376**: 307–312.
- Bressler, S. L. (1987). Relation of olfactory bulb and cortex. I. Spatial variation of bulbo-cortical interdependence, *Brain Research* **409**: 285–293.
- Bressler, S. L. (1990). The gamma wave: A cortical information carrier?, *Trends Neurosci.* **13**: 161–162.
- Brown, T. H., Kairiss, E. W. and Keenan, C. L. (1990). Hebbian synapses: Biophysical mechanisms and algorithms, *Ann. Rev. Neurosci.* **13**: 475–511.

- Canavier, C. C., Baxter, D. A., Clark, J. W. and Byrne, J. H. (1993). Nonlinear dynamics of a model neuron provide a novel mechanism for transient synaptic inputs to produce long-term alterations of postsynaptic activity, *J. Neurophysiol.* **69**: 2252–2257.
- Canavier, C. C., Clark, J. W. and Byrne, J. H. (1991). Simulation of the bursting activity of neuron R15 in *Aplysia*: Role of ionic currents, calcium balance, and modulatory transmitters, *J. Neurophysiol.* **66**: 2107–2124.
- Chomsky, N. (1957). *Syntactic Structures*, Mouton, The Hague.
- Churchland, P. S. and Sejnowski, T. J. (1988). Perspectives on cognitive neuroscience, *Science* **242**: 741–745.
- Coggeshall, R. E. (1967). A light and electron microscope study of the abdominal ganglion of *Aplysia californica*, *J. Neurophysiol.* **30**: 1263–1287.
- Cohen, A. (1992). The role of heterarchical control in the evolution of the central pattern generator for locomotion, *Brain, Behavior and Evolution* **40**: 112–124.
- Cohen, A. and Kiemel, T. (1993). Intersegmental coordination: Lessons from modeling systems of coupled non-linear oscillators, *American Zoology* **33**: 54–65.
- Cohen, A., Ermentrout, G. B., Kiemel, T., Kopell, N., Sigvardt, K. A. and Williams, T. L. (1992). Modeling of intersegmental coordination in the lamprey central pattern generator for locomotion, *Trends Neurosci.* **15**: 434–438.
- Cole, K. (1949). Dynamic electrical characteristics of the squid axon membrane, *Arch. Sci. Physiol.* **3**: 253–258.
- Cole, K. (1968). *Membranes, Ions, and Impulses: A Chapter of Classical Biophysics*, Univ. of California Press, Berkeley.
- Cole, K. and Curtis, H. (1939). Electric impedance of the squid giant axon during activity, *J. Gen. Physiol.* **22**: 649–670.
- Collins, R. and Jefferson, D. (1991). Selection in massively parallel genetic algorithms, *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kauffman, San Mateo, CA, pp. 249–256.
- Connor, J. A. and Stevens, C. F. (1971a). Inward and delayed outward membrane currents in isolated neural somata under voltage clamp, *J. Physiol.* **213**: 1–20.
- Connor, J. A. and Stevens, C. F. (1971b). Prediction of repetitive firing behavior from voltage clamp data on an isolated neurone soma, *J. Physiol.* **213**: 31–53.

- Connor, J. A. and Stevens, C. F. (1971c). Voltage clamp studies of a transient outward membrane current in gastropod neural somata, *J. Physiol.* **213**: 21–30.
- Constanti, A. and Galvan, M. (1983). Fast-inward rectifying current accounts for anomalous rectification in olfactory cortex neurones, *J. Physiol. (London)* **385**: 153–178.
- Constanti, A. and Sim, J. A. (1987). Calcium-dependent potassium conductance in guinea-pig olfactory cortex neurones *in vitro*, *J. Physiol. (London)* **387**: 173–194.
- Constanti, A., Galvan, M., Franz, P. and Sim, J. A. (1985). Calcium-dependent inward currents in voltage clamped guinea-pig olfactory cortex neurones, *Pfälzers Arch.* **404**: 259–265.
- Crick, F. and Koch, C. (1990). Towards a neurobiological theory of consciousness, *Sem. Neurosci.* **2**: 263–275.
- Curtis, H. and Cole, K. (1940). Membrane action potentials from the squid giant axon, *J. Cell. Comp. Physiol.* **15**: 147–157.
- De Schutter, E. and Bower, J. M. (1994a). An active membrane model of the cerebellar Purkinje cell I. Simulation of current clamps in slice, *J. Neurophysiol.* **71**: 375–400.
- De Schutter, E. and Bower, J. M. (1994b). An active membrane model of the cerebellar Purkinje cell II. Simulation of synaptic responses, *J. Neurophysiol.* **71**: 401–419.
- De Schutter, E. and Bower, J. M. (1994c). Responses of cerebellar Purkinje cells are independent of the dendritic location of granule cell synaptic inputs, *Proc. Natl. Acad. Sci. (US)* **91**: 4736–4740.
- De Schutter, E. and Smolen, P. (1997). Calcium dynamics in large neuronal models, in C. Koch and I. Segev (eds.), *Methods in Neuronal Modeling: From synapses to networks*, second edn., MIT Press, Cambridge, MA, pp. 63–96. In press.
- Devor, M. (1976). Fiber trajectories of olfactory bulb afferents in hamster, *J. Comp. Neurol.* **166**: 31–48.
- Døskeland, S. O. and Øgreid, D. (1984). Characterization of the interchain and intrachain interactions between the binding sites of the free regulatory moiety of protein kinase I, *J. Biol. Chem.* **259**: 2291–2301.
- Eckman, F. H. and Bower, J. M. (eds.) (1993). *Computation and Neural Systems*, Kluwer Academic Publishers, Boston.
- Ermentrout, G. B. and Kopell, N. (1990). Oscillator death in systems of coupled neural oscillators, *SIAM Journal of Applied Mathematics* **50**: 125–146.

- Fatt, P. and Katz, B. (1953). The effect of inhibitory nerve impulses on a crustacean muscle fiber, *J. Physiol. (London)* **121**: 374–389.
- Frazier, W. T., Kandel, E. R., Kupfermann, I., Waziri, R. and Coggeshall, R. E. (1967). Morphological and functional properties of identified neurons in the abdominal ganglion of *Aplysia californica*, *J. Neurophysiol.* **30**: 1288–1351.
- Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V. (1994). *PVM: Parallel Virtual Machine*, MIT Press, Cambridge, MA. Also see http://www.epm.ornl.gov/pvm/pvm_home.html.
- Getting, P. A. (1989). Reconstruction of small neural networks, in C. Koch and I. Segev (eds.), *Methods in Neuronal Modeling*, MIT Press, Cambridge, MA, chapter 6, pp. 171–194.
- Gilman, A. G. (1987). G proteins: Transducers of receptor-generated signals, *Ann. Rev. Biochem.* **56**: 615–649.
- Goddard, N. and Hood, G. (1996). Parallel GENESIS at PSC, <http://www.psc.edu/general/software/packages/genesis>.
- Goddard, N. H., Fanty, M. A. and Lynne, K. (1987). The Rochester connectionist simulator, *Technical Report* 233, University of Rochester, Computer Science Department, Rochester, NY.
- Gorman, A. L. F. and Hermann, A. (1982). Quantitative differences in the currents of bursting and beating moluscan pace-maker neurones, *J. Physiol.* **333**: 681–699.
- Gorman, A. L. F. and Thomas, M. V. (1980). Potassium conductance and internal calcium accumulation in a molluscan neurone, *J. Physiol.* **308**: 287–313.
- Gorman, A. L. F., Hermann, A. and Thomas, M. V. (1982). Ionic requirements for membrane oscillations and their dependence on the calcium concentration in a molluscan pace-maker neurone, *J. Physiol.* **327**: 185–217.
- Gray, C. M., Konig, P., Engel, A. K. and Singer, W. (1989). Oscillatory responses in cat visual-cortex exhibit inter-columnar synchronization which reflects global stimulus properties, *Nature* **338**: 334–337.
- Grillner, S. (1974). On the generation of locomotion in the spinal dogfish, *Exp. Brain Res.* **20**: 459–470.
- Grillner, S. (1981). Control of locomotion in bipeds, tetrapods, and fish, in V. B. Brooks (ed.), *Handbook of Physiology: The Nervous System II*, American Physiology Society, Bethesda, MD, chapter 26, pp. 1179–1236.

- Haberly, L. B. (1973a). Summed potentials evoked in opossum prepyriform cortex, *J. Neurophysiol.* **36**: 775–788.
- Haberly, L. B. (1973b). Unitary analysis of opossum prepyriform cortex, *J. Neurophysiol.* **36**: 762–774.
- Haberly, L. B. (1978). Application of collision testing to investigate properties of association axons originating from single cells in the piriform cortex of the rat, *Soc. Neurosci. Abst.* **4**: 75.
- Haberly, L. B. (1983). Structure of the piriform cortex of the opossum. I. description of neuron types with golgi methods, *J. Comp. Neurol.* **219**: 448–460.
- Haberly, L. B. (1985). Neuronal circuitry in olfactory cortex: Anatomy and functional applications, *Chemical Senses* **10**: 219–238.
- Haberly, L. B. (1990). Olfactory cortex, in G. M. Shepherd (ed.), *The Synaptic Organization of the Brain*, Oxford University Press, New York, chapter 10, pp. 317–345.
- Haberly, L. B. and Bower, J. M. (1984). Analysis of association fiber pathway in piriform cortex with intracellular recording and staining techniques, *J. Neurophysiol.* **51**: 90–112.
- Haberly, L. B. and Bower, J. M. (1989). Olfactory cortex – model circuit for study of associative memory, *Trends Neurosci.* **12**: 258–264.
- Haberly, L. B. and Price, J. L. (1978). Association and commissural fiber system of the olfactory cortex of the rat. I. System originating in the piriform cortex and adjacent areas, *J. Comp. Neurol.* **178**: 711–740.
- Harris-Warrick, R. M., Marder, E., Selverston, A. I. and Moulins, M. (eds.) (1992). *Dynamic Biological Networks*, MIT Press, Cambridge, MA.
- Hasselmo, M. E. (1993). Acetylcholine and learning in a cortical associative memory, *Neural Comp.* **5**: 22–34.
- Hasselmo, M. E. and Bower, J. M. (1992). Cholinergic suppression specific to intrinsic not afferent fiber synapses in rat piriform (olfactory) cortex, *J. Neurophysiol.* **67**: 1222–1229.
- Hasselmo, M. E. and Bower, J. M. (1993). Acetylcholine and memory, *Trends Neurosci.* **16**: 218–222.
- Hasselmo, M. E., Anderson, B. P. and Bower, J. (1992). Cholinergic modulation of cortical associative memory function, *J. Neurophysiol.* **67**: 1230–1246.

- Hasselmo, M. E., Barkai, E., Horwitz, G. and Bergman, R. (1994). Modulation of neuronal adaptation and cortical associative memory function, in F. H. Eeckman (ed.), *Computation in Neurons and Neural Systems*, Kluwer Academic Publishers, Norwell, MA, pp. 287–292.
- Hasselmo, M. E., Wilson, M. A., Anderson, B. P. and Bower, J. M. (1990). Associative memory function in piriform (olfactory) cortex – computational modeling and neuropsychopharmacology, *Cold-S-Harb* **55**: 599–610.
- Hebb, D. O. (1949). *The Organization of Behavior*, Wiley, New York.
- Hille, B. (1984). *Ionic Channels of Excitable Membranes*, Sinauer, Sunderland, MA.
- Hille, B. (1992). *Ionic Channels of Excitable Membranes*, second edn., Sinauer Associates Inc., Sunderland, MA.
- Hines, M. (1984). Efficient computation of branched nerve equations, *Int. J. Bio-Med. Comput.* **15**: 69–79.
- Hodgkin, A. (1976). Chance and design in electrophysiology: An informal account of certain experiments on nerve carried out between 1934 and 1952, *J. Physiol. (London)* **263**: 1–21.
- Hodgkin, A. and Huxley, A. (1939). Action potentials recorded from inside a nerve fibre, *Nature* **144**: 710–711.
- Hodgkin, A. and Huxley, A. (1952a). Currents carried by sodium and potassium ions through the membrane of the giant axon of *Loligo*, *J. Physiol. (London)* **116**: 449–472.
- Hodgkin, A. and Huxley, A. (1952b). The components of membrane conductance in the giant axon of *Loligo*, *J. Physiol. (London)* **116**: 473–496.
- Hodgkin, A. and Huxley, A. (1952c). The dual effect of membrane potential on sodium conductance in the giant axon of *Loligo*, *J. Physiol. (London)* **116**: 497–506.
- Hodgkin, A. and Huxley, A. (1952d). A quantitative description of membrane current and its application to conduction and excitation in nerve, *J. Physiol. (London)* **117**: 500–544.
- Hodgkin, A. and Katz, B. (1949). The effect of sodium ions on the electrical activity of the giant axon of the squid, *J. Physiol. (London)* **108**: 37–77.
- Hodgkin, A., Huxley, A. and Katz, B. (1952). Measurement of current-voltage relations in the membrane of the giant axon of *Loligo*, *J. Physiol. (London)* **116**: 424–448.

- Holmes, W. R. and Levy, W. B. (1990). Insights into associative long-term potentiation from computational models of NMDA receptor-mediated calcium influx and intracellular calcium concentration changes, *J. Neurophysiol.* **63**: 1148–1168.
- Hooper, S. L. and Marder, E. (1987). Modification of the lobster pyloric rhythm by the peptide proctolin, *J. Neurosci.* **7**: 2097–2112.
- Huguenard, J. and McCormick, D. (1994). *Electrophysiology of the Neuron: An Interactive Tutorial*, Oxford University Press, New York.
- Jack, J. J. B., Noble, D. and Tsien, R. W. (1975). *Electric Current Flow in Excitable cells*, Calderon Press, Oxford.
- Jaeger, D., E. De Schutter, E. and Bower, J. M. (1997). The role of synaptic and voltage-gated currents in the control of Purkinje cell spiking: A modeling study, *J. Neurosci.* **17**: 91–106.
- Jahr, C. E. and Stevens, C. F. (1990). A qualitative description of NMDA receptor channel kinetic behavior, *J. Neurosci.* **10**: 1830–1837.
- Johnston, D. and Wu, S. M.-S. (1995). *Foundations of Cellular Neurophysiology*, MIT Press, Cambridge, MA.
- Kandel, E. R. (1976). *Cellular Basis of Behavior*, W. F. Freeman, San Francisco.
- Kandel, E. R., Schwartz, J. H. and Jessell, T. M. (1991). *Principles of Neural Science*, third edn., Appleton and Lange, Norwalk, CN.
- Kanter, E. D. and Haberly, L. B. (1990). NMDA-dependent induction of long-term potentiation in afferent and association fiber systems of piriform cortex *in vitro*, *Brain Res.* **525**: 175–179.
- Katz, B. and Miledi, R. (1969). Tetrodotoxin-resistant electrical activity in presynaptic terminals, *J. Physiol.* **192**: 407–436.
- Ketchum, K. L. and Haberly, L. B. (1991). Fast oscillations and dispersive propagation in olfactory cortex and other cortical areas: A functional hypothesis, in J. Davis and H. Eichenbaum (eds.), *Olfaction: A Model System for Computational Neuroscience*, MIT Press, Cambridge, MA, chapter 3, pp. 69–100.
- Koch, C. and Poggio, T. (1987). Biophysics of computation: Neurons, synapses, and membranes, in G. M. Edelman, W. E. Gall and W. M. Cowan (eds.), *Synaptic Function*, W. M. Wiley, New York, pp. 637–698.

- Koch, C. and Zador, T. (1993). The function of dendritic spines: Devices subserving biochemical rather than electrical compartmentalization, *J. Neuroscience* **13**: 413–422.
- Koch, C., Poggio, T. and Torre, V. (1982). Retinal ganglion cells: A functional interpretation of dendritic morphology, *Phil. Trans. R. Soc. Lond. (Biol.)* **298**: 227–264.
- Kopell, N. and Ermentrout, G. B. (1986). Symmetry and phaselocking in chains of weakly coupled oscillators, *Comm. Pure and Appl. Math.* **39**: 623–660.
- Kramer, R. H. and Zucker, R. S. (1985a). Calcium-dependent inward current in *Aplysia* bursting pace-maker neurones, *J. Physiol.* **362**: 107–130.
- Kramer, R. H. and Zucker, R. S. (1985b). Calcium-induced inactivation of calcium current causes the inter-burst hyperpolarization of *Aplysia* bursting neurones, *J. Physiol.* **362**: 131–160.
- Larson, J., Wong, D. and Lynch, G. (1986). Patterned stimulation at the theta frequency is optimal for the induction of hippocampal long-term potentiation, *Brain Res.* **368**: 347–350.
- Lauffenburger, D. A. and Linderman, J. J. (1993). *Models for binding, trafficking, and signaling*, Oxford University Press, New York.
- Laurent, G. (1993). A dendritic gain control mechanism in axonless neurons of the locust, *Schistocerca americana*, *J. Physiol. (London)* **470**: 45–54.
- Leigh, J., De Schutter, E., Lee, E., Bhalla, U. S., Bower, J. M. and DeFanti, T. A. (1983). Realistic modeling of brain structures with remote interaction between simulations of an inferior olfactory neuron and a cerebellar Purkinje cell, *Proceedings of the SCS Simulations Multiconference*, Arlington, VA.
- Levitzki, A. (1984). *Receptors: A quantitative approach*, Benjamin-Cummings, Menlo Park, CA.
- Levy, W. B. (1996). A sequence predicting CA3 is a flexible associator that learns and uses context to solve hippocampal-like tasks, *Hippocampus* **6**: 579–590.
- Llinás, R. (1988). The intrinsic electrophysiological properties of mammalian neurons: Insights into central nervous system function, *Science* **242**: 1654–1664.
- Llinás, R., Steinberg, I. Z. and Walton, K. (1976). Presynaptic calcium currents and their relation to synaptic transmission: Voltage clamp study in squid giant synapse and theoretical model for calcium gate, *Proc. Natl. Acad. Sci. USA* **73**: 2918–2922.

- Lundberg, A. and Phillips, C. G. (1973). T. Graham Brown's film on locomotion in decerebrate cat, *J. Physiol.* **231**: 90.
- MacGregor, R. J. (1987). *Neural and Brain Modeling*, Academic Press, San Diego.
- Macrides, F., Eichenbaum, H. B. and Forbes, W. B. (1982). Temporal relationship between sniffing and the limbic theta-rhythm during odor discrimination reversal-learning, *J. Neurosci.* **2**: 1705–1717.
- Madison, D. V. and Nicoll, R. A. (1982). Noradrenaline blocks accommodation of pyramidal cell discharge in the hippocampus, *Nature* **299**: 636–638.
- Marder, E. and Meyrand, P. (1989). Chemical modulation of an oscillatory neural circuit, in W. Jacklet (ed.), *Neuronal and Cellular Oscillators*, Marcel Dekker, Inc., New York, chapter 11, pp. 317–338.
- Marmont, G. (1949). Studies on the axon membrane. I. A new method., *J. Cell. Comp. Physiol.* **34**: 351–382.
- Marr, D. (1982). *Vision*, W. H. Freeman, San Francisco.
- Mascagni, M. V. (1989). Numerical methods for neuronal modeling, in C. Koch and I. Segev (eds.), *Methods in Neuronal Modeling*, MIT Press, Cambridge, MA, chapter 13, pp. 439–484.
- McClelland, J. and Goddard, N. (1996). Considerations arising from a complementary learning systems perspective on hippocampus and neocortex, *Hippocampus* **6**: 654–665.
- McCormick, D. A. (1990). Membrane properties and neurotransmitter actions, in G. M. Shepherd (ed.), *Synaptic Organization of the Brain*, third edn., Oxford University Press, New York, chapter 2, pp. 32–66.
- McCormick, D. A., Huguenard, J. and Strowbridge, B. W. (1992). Determination of state-dependent processing in thalamus by single neuron properties and neuromodulators, in T. McKenna, J. Davis and S. Zornetzer (eds.), *Single Neuron Computation*, Academic Press, San Diego, chapter 10, pp. 259–290.
- McKenna, T., Davis, J. and Zornetzer, S. F. (eds.) (1992). *Single Neuron Computation*, Academic Press, San Diego.
- Mel, W. B. (1993). Synaptic integration in an excitable dendritic trees, *J. Neurophys.* **70**: 1086–1101.
- Michaelis, L. and Menten, M. L. (1913). *Biochem. Z.* **49**: 333.

- Moczydlowski, E. and Latorre, R. (1983). Gating kinetics of Ca^{2+} -activated K^+ channels from rat muscle incorporated into planar lipid bilayers: Evidence for two voltage-dependent Ca^{2+} binding reactions, *J. Gen. Physiol.* **82**: 511–542.
- Murray, J. D. (1989). *Mathematical Biology*, Springer-Verlag, New York.
- Nelson, M. E. and Bower, J. M. (1990). Brain maps and parallel computers, *Trends Neurosci.* **13**: 403–408.
- Nelson, M., Furmanski, W. and Bower, J. M. (1989). Simulating neurons and neuronal networks on parallel computers, in C. Koch and I. Segev (eds.), *Methods in Neuronal Modeling*, MIT Press, Cambridge, MA, chapter 12, pp. 397–438.
- Nicholls, J. G., Martin, A. R. and Wallace, B. G. (1992). *From Neuron to Brain*, third edn., Sinauer Associates, Inc., Sunderland, MA.
- Nishizuka, Y. (1992). Intracellular signaling by hydrolysis of phospholipids and activation of protein kinase C, *Science* **258**: 607–614.
- Nunez, P. L. (1981). *Electric Fields of the Brain: The Neurophysics of EEG*, Oxford University Press, Oxford.
- Otmakhov, N., Shirke, A. M. and Malinow, R. (1993). Measuring the impact of probabilistic transmission on neuronal output, *Neuron* **10**: 1101–1111.
- Pearson, K. (1976). The control of walking, *Scientific American* **235**: 72–86.
- Pellionisz, A. and Llinás, R. (1977). A computer model of cerebellar Purkinje cells, *Neuroscience* **2**: 37–48.
- Pinski, P. F. and Rinzel, J. (1994). Intrinsic and network rhythmogenesis in a reduced Traub model for CA3 neurons, *J. Comput. Neurosci.* **1**: 39–60.
- Press, W. H., Flannery, B. P., Teukolsky, S. A. and Vetterling, W. T. (1986). *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, Cambridge.
- Price, J. L. (1973). An autoradiographic study of complementary laminar patterns of termination of afferent fibers to the olfactory cortex, *J. Comp. Neurol.* **150**: 87–108.
- Protopapas, A. and Bower, J. M. (1994). Sensitivity in the response of piriform pyramidal cells to fluctuations in synaptic timing, in F. H. Eeckman (ed.), *Computation in Neurons and Neural Systems*, Kluwer Academic Publishers, Norwell, MA, pp. 185–190.
- R. J. Butera, J., J. W. Clark, J. and Byrne, J. H. (1996). Dissection and reduction of a modeled bursting neuron, *J. Comp. Neurosci.* **3**: 199–223.

- R. J. Butera, J., J. W. Clark, J., Canavier, C. C., Baxter, D. A. and Byrne, J. H. (1995). Analysis of the effects of modulatory agents on a modeled bursting neuron: Dynamic interactions between voltage and calcium dependent systems, *J. Comp. Neurosci.* **2**: 19–44.
- Rall, W. (1959). Branching dendritic trees and motoneuron membrane resistivity, *Exp. Neurol.* **1**: 491–527.
- Rall, W. (1964). Theoretical significance of dendritic trees for neuronal input-output relations, in R. Reiss (ed.), *Neuronal Theory and Modeling*, Stanford University Press, Stanford, CA, pp. 73–97.
- Rall, W. (1967). Distinguishing theoretical synaptic potentials computed for different soma-dendritic distribution of synaptic inputs, *J. Neurophysiol.* **30**: 1138–1168.
- Rall, W. (1969). Time constant and electrotonic length of membrane cylinders and neurons, *Biophys. J.* **9**: 1483–1508.
- Rall, W. (1977). Cable theory for neurons, in E. R. Kandel, J. M. Brookhardt and V. B. Mountcastle (eds.), *Handbook of Physiology: The Nervous System*, Vol. 1, Williams and Wilkins, Baltimore, chapter 3, pp. 39–98.
- Rall, W. (1989). Cable theory for dendritic neurons, in C. Koch and I. Segev (eds.), *Methods in Neuronal Modeling*, MIT Press, Cambridge, MA, chapter 2, pp. 9–62.
- Rall, W. and Rinzel, J. (1973). Branch input resistance and steady state attenuation for input to one branch of a dendritic neuron model, *Biophys. J.* **13**: 648–688.
- Rall, W. and Segev, I. (1987). Functional possibilities for synapses on dendrites and on dendritic spines, in G. M. Edelman, E. E. Gall and W. M. Cowan (eds.), *Synaptic Function*, Wiley, New York, pp. 605–636.
- Rall, W. and Shepherd, G. M. (1968). Theoretical reconstruction of field potentials and dendrodendritic synaptic interactions in the olfactory bulb, *J. Neurophysiol.* **31**: 884–915.
- Ranck, J. B. (1973). Studies on single neurons in dorsal hippocampal formation and septum in unrestrained rats. I. behavioral correlates and firing repertoires, *Exp. Neurol.* **41**: 462–531.
- Rand, R. H., Cohen, A. H. and Holmes, P. J. (1988). Systems of coupled oscillators as models of central pattern generators, in A. H. Cohen (ed.), *Neural Control of Rhythmic Movements in Vertebrates*, Wiley, New York, chapter 9, pp. 333–367.

- Rapp, M., Yarom, Y. and Segev, I. (1992). The impact of parallel fiber background activity on the cable properties of cerebellar Purkinje cells, *Neural Computation* **4**: 518–533.
- Regehr, W. G., Connor, J. A. and Tank, D. W. (1989). Optical imaging of calcium accumulation in hippocampal pyramidal cells during synaptic activation, *Nature* **341**: 533–536.
- Rinzel, J. (1990). Electrical excitability of cells, theory and experiment: Review of the Hodgkin-Huxley foundation and an update, in M. Mangel (ed.), *Bull. Math. Biology: Classics of Theoretical Biology*, Vol. 52, pp. 5–23.
- Rinzel, J. and Ermentrout, G. B. (1989). Analysis of neural excitability and oscillations, in C. Koch and I. Segev (eds.), *Methods in Neuronal Modeling*, MIT Press, Cambridge, MA, chapter 5, pp. 135–169.
- Rinzel, J. and Lee, Y. S. (1987). Dissection of a model for neuronal parabolic bursting, *J. Math. Biol.* **25**: 653–675.
- Rinzel, J. and Rall, W. (1974). Transient response in a dendritic neuron model for current injected at one branch, *Biophys. J.* **14**: 759–790.
- Sala, F. and Hernández-Cruz, A. (1987). Calcium diffusion modeling in a spherical neuron, *Biophys. J.* **57**: 313–324.
- Santamaria, F. and Bower, J. M. (1997). A realistic cerebellar network: The temporal and spatial, direct and indirect influences of granule cell axons on Purkinje cells, *Soc. Neurosci. Abst.* In Press.
- Satou, M., Mori, K., Tazawa, Y. and Takagi, S. F. (1982). Long lasting disinhibition in pyriform cortex of the rabbit, *J. Neurophysiol.* **48**: 1157–1163.
- Schwob, J. E. and Price, J. L. (1978). The cortical projections of the olfactory bulb: Development in fetal and neonatal rats correlated with quantitative variations in adult rats, *Brain Res.* **151**: 369–374.
- Segev, I. (1992). Single neurone models: Oversimple, complex and reduced, *Trends Neurosci.* **15**: 414–421.
- Segev, I. (1995). Denritic processing, in M. A. Arbib (ed.), *The Handbook of Brain Theory and Neural Networks*, MIT Press, Cambridge, MA.
- Segev, I. and Parnas, I. (1983). Synaptic integration mechanisms: A theoretical and experimental investigation of temporal postsynaptic interactions between excitatory and inhibitory inputs, *Biophys. J.* **41**: 41–50.

- Segev, I. and Rall, W. (1988). Computational study of an excitable dendritic spine, *J. Neurophysiol.* **60**: 499–523.
- Segev, I., Fleshman, J. W. and Burke, R. E. (1989). Compartmental models of complex neurons, in C. Koch and I. Segev (eds.), *Methods in Neuronal Modeling*, MIT Press, Cambridge, MA, chapter 3, pp. 63–96.
- Segev, I., Fleshman, J. W., Miller, J. P. and Bunow, B. (1985). Modeling the electrical behavior of anatomically complex neurons using a network program: Passive membrane, *Biol. Cybern.* **53**: 27–40.
- Segev, I., Rinzel, J. and Shepherd, G. H. (eds.) (1995). *The Theoretical Foundation of Dendritic Function: Selected Papers by Wilfrid Rall with Commentaries*, MIT Press, Cambridge, MA.
- Selverston, A. I. (ed.) (1985). *Model Neural Networks and Behavior*, Plenum Press, New York.
- Shepherd, G. M. (1990). *The Synaptic Organization of the Brain*, third edn., Oxford University Press, New York.
- Shepherd, G. M. (1994). *Neurobiology*, third edn., Oxford University Press, New York.
- Shepherd, G. M. and Greer (1990). Olfactory bulb, in G. M. Shepherd (ed.), *Synaptic Organization of the Brain*, third edn., Oxford University Press, New York, chapter 5, pp. 149; 154–144.
- Smith, S. J. and Thompson, S. H. (1987). Slow membrane currents in bursting pace-maker neurones of *Tritonia*, *J. Physiol.* **382**: 425–448.
- Staubli, U. and Lynch, G. (1987). Stable hippocampal long-term potentiation elicited by ‘theta’ pattern stimulation., *Brain Res.* **435**: 227–234.
- Strumwasser, F. (1965). The demonstration and manipulation of a circadian rhythm in a single neuron, in J. Aschoff (ed.), *Circadian Clocks*, Noth-Holland, Amsterdam, pp. 442–462.
- Strumwasser, F. (1968). Membrane and intracellular mechanisms governing endogenous activity in neurons, in F. D. Carlson (ed.), *Physiological and Biochemical Aspects of Nervous Integration*, Englewood Cliffs, N. J., pp. 329–341.
- Stuart, G. J. and Sakmann, B. (1994). Active propagation of somatic action potentials into neocortical pyramidal cell dendrites, *Nature* **367**: 69–72.

- Szekely, G. (1968). Development of limb movements: Embryological, physiological and model studies, in G. E. W. Wolstenholme and M. O'Connor (eds.), *Ciba Foundation Symposium on Growth of the Nervous System*, Churchill, London, pp. 77–93.
- Thompson, S. H. (1977). Three pharmacologically distinct potassium channels in molluscan neurones, *J. Physiol.* **265**: 465–488.
- Thompson, S., Smith, S. J. and Johnson, J. W. (1986). Slow outward tail currents in molluscan bursting pacemaker neurons: Two components differing in temperature sensitivity, *J. Neurosci.* **6**: 3169–3176.
- Traub, R. D., Jeffereys, J. G. R., Miles, R., Whittington, M. A. and Tóth, K. (1994). A branching dendritic model of a rodent CA3 pyramidal neurone, *J. Physiol. (London)* **481**: 79–95.
- Traub, R. D., Wong, R. K. S., Miles, R. and Michelson, H. (1991). A model of a CA3 hippocampal neuron incorporating voltage-clamp data on intrinsic conductances, *J. Neurophysiol.* **66**: 635–650.
- Tseng, G. and Haberly, L. B. (1986). A synaptically mediated K⁺ potential in olfactory cortex: Characterization and evidence for interneuronal origin, *Soc. Neurosci. Abst.* **12**: 667.
- Tseng, G. and Haberly, L. B. (1989). Deep neurons in piriform cortex. II. Membrane properties that underlie unusual synaptic responses, *J. Neurophysiol.* **62**: 386–400.
- Vanier, M. C. and Bower, J. M. (1993). Differential effects of norepinephrine on synaptic transmission in layers Ia and Ib of rat olfactory cortex, in F. H. Eeckman and J. M. Bower (eds.), *Computation and Neural Systems*, Kluwer Academic Publishers, Norwell, MA, pp. 267–271.
- Vanier, M. C. and Bower, J. M. (1996). A comparison of automated parameter-searching methods for neural models, in J. M. Bower (ed.), *Computational Neuroscience: Trends in Research 1995*, Academic Press, New York, pp. 477–482.
- White, E. L. (1989). *Cortical circuits: Synaptic organization of the cerebral cortex — Structure, function and theory*, Birkhäuser, Boston.
- Wilson, C. J. (1992). Dendritic morphology, inward rectification and the functional properties of neostriatal neurons, in T. McKenna, J. Davis and S. Zornetzer (eds.), *Single Neuron Computation*, Academic Press, San Diego, pp. 141–172.
- Wilson, M. A. (1990). *CIT Thesis*, PhD thesis, California Institute of Technology, Pasadena.

- Wilson, M. A. and Bower, J. M. (1989). The simulation of large scale neural networks, in C. Koch and I. Segev (eds.), *Methods in Neuronal Modeling*, MIT Press, Cambridge, MA, chapter 9, pp. 291–333.
- Wilson, M. A., McNaughton, B. L. and Stengel, K. (1992). Large scale parallel recording of multiple single unit activity in the hippocampus and parietal cortex of the behaving rat, *Soc. eurosci. Abst.* **18**: 1216.
- Wilson, M. and Bower, J. M. (1988). A computer simulation of olfactory cortex with functional implications for storage and retrieval of olfactory information, in D. Anderson (ed.), *Neural Information Processing Systems*, American Institute of Physics, New York, pp. 114–126.
- Wilson, M. and Bower, J. M. (1991). A computer simulation of oscillatory behavior in primary visual cortex, *Neural Computation* **3**: 498–509.
- Wilson, M. and Bower, J. M. (1992). Cortical oscillations and temporal interactions in a computer simulation of piriform cortex, *J. Neurophysiol.* **67**: 981–995.
- Wilson, W. A. and Wachtel, H. (1974). Negative resistance characteristic essential for the maintenance of slow oscillations in bursting neurons, *Science* **186**: 932–934.
- Yamada, W. M., Koch, C. and Adams, P. R. (1989). Multiple channels and calcium dynamics, in C. Koch and I. Segev (eds.), *Methods in Neuronal Modeling*, MIT Press, Cambridge, MA, chapter 4, pp. 97–133.
- Zador, A., Koch, C. and Brown, T. H. (1990). Biophysical model of a Hebbian receptor channel kinetic behavior, *Proc. Natl. Acad. Sci. USA* **10**: 6718–6722.

Index of GENESIS Commands

addaction, 238
addfield, 237, 316
addmsg, 209
addobject, 239
async, 358, 376

barrier, 354, 356, 376
barrierall, 369, 376

call, 305
ce, 207
check, 208
clearthread, 375
clearthreads, 371, 375
copy, 284, 309, 397
create, 206
createmap, 284, 365, 371

debug, 224
delete, 206
deletemsg, 210
disable, 258
duplicatetable, 310

echo, 217
enable, 258

end, 216, 237, 285
exists, 305
exit, 211

file2tab, 307
findsolvefield, 339, 340, 345
floatformat, 217, 299
for, 211, 285, 312
foreach, 285

getclock, 222
getfield, 229
getsyncount, 297
getsyndest, 298
getsynindex, 298
getsynsrc, 298

help, 206

if, 237
if-else, 249
include, 231
isa, 239

le, 207
listcommands, 205

listglobals, 224
listobjects, 205
move, 397
mynode, 355, 375
mypvmid, 375
mytotalnode, 369, 375
myzone, 355, 375
nnodes, 375
npvmcpu, 375
ntotalnodes, 375
nzones, 375
paroff, 361, 375
paron, 360, 375
pixflags, 388
planarconnect, 288
planardelay, 292
planarweight, 294
pope, 207
position, 285
pushe, 207
pwe, 207
quit, 211
raddmsg, 360, 376
randseed, 296
readcell, 255
readsolve, 339
reset, 208, 234
restore, 330
rshowmsg, 376
rvolumeconnect, 365, 376
rvolumedelay, 366, 376
rvolumeweight, 366, 376
save, 330
scaletabchan, 309
setclock, 222
setfield, 208
setfieldprot, 237
setmethod, 337
setupalpha, 312
setupghk, 314
setupNaCa, 314
showfield, 207
showmsg, 210
showobject, 205
showstat, 331
silent, 342
step, 208, 223
syndelay, 293
threadsoff, 375
threadsone, 375
tweakalpha, 306
tweaktau, 306
useclock, 222
volumeconnect, 291
volumedelay, 293
volumeweight, 295
waiton, 359, 376
while, 285
xcolorscale, 390
xhide, 209
xshow, 209

Index of GENESIS Objects

asc_file, 374
axon, 245

Ca_concen, 315
channelC2, 245
compartment, 205
concchan, 181

ddsyn, 326
difbuffer, 314
difshell, 314
disk_out, 375

enz, 181

fixbuffer, 314

hebbsynchan, 252
hh_channel, 229
hsolve, 180, 337

ksolve, 180

Mg_block, 324
mmpump, 314

nernst, 317
neutral, 206

par_asc_file, 374, 376
par_disk_out, 375, 376
pool, 180, 181
postmaster, 331, 376
pulsegen, 212

randomspike, 246
reac, 181

script_out, 251
spikegen, 248
symcompartment, 205
synchan, 244

tab2Dchannel, 229, 321
tabchannel, 229, 301–312
tabcurrent, 314
tabgate, 229, 320
table, 318, 320, 391
table2D, 320

vdep_channel, 229, 318
vdep_gate, 229, 321

- xbutton**, 210, 226
xcell, 394
xcoredraw, 396
xdialog, 228
xdraw, 385
xform, 209
xgif, 389
xgraph, 209, 247
xlabel, 227
xpix, 387
xshape, 385, 389
xsphere, 388
xtoggle, 249
xtree, 394, 396
xvar, 390
xview, 374, 393

Index

- action potential, 29, 32, 47, 102
alpha function, *see* conductance, alpha function form
anode break, 49, 103, 128
Aplysia, 99
associative memory, 150
axial resistance, 11, 58, 218
axon, 12, 247, 248
axon hillock, 7
- BABEL, vii, 18, 409
beater, 99
burster, 99, 112
- cable
 voltage attenuation, 59, 71
 voltage decay, 60
cable boundary condition
 clamped to rest, 59, 64
 leaky end, 59, 64
 sealed end, 59, 64
cable equation, 58
cable theory, 56
calcium concentration, 117, 119, 120, 124, 314, 316
- calcium-activated potassium conductance, *see* conductance, C-current
cell parameter file, *see* GENESIS, cell descriptor file
central pattern generator, 131
 invertebrate, 132
 vertebrate, 132
channel, *see* conductance
 C_M , *see* specific, membrane capacitance
 C_m , *see* membrane capacitance
compartment, 10, 66, 203
compartmental modeling, *see* modeling, compartmental
computational neuroscience, 3, 121
concentration-effect curve, 178, 185
conductance, 101
 A-current, 110, 111, 117
 AHP Current, 317
 AHP current, 122
 alpha function form, 13, 88
 B-current, 107–109, 111, 116, 302
 C-current, 107, 109, 111, 116, 120, 122, 318
 delayed potassium, 39, 104, 116
 dual exponential form, 89

- fast sodium, 45, 103, 116
high threshold calcium, 104, 105, 116, 122
leakage, 12
low threshold calcium, 107, 276
muscarinic potassium, 276
NMDA, 73, 85, 324
synaptic, 80, 85
conductance density, 115, 116
connection, *see* synaptic coupling
coupled oscillators, 134
 chains of, 140
CPG, *see* central pattern generator
current clamp, 47
cytoplasm resistivity, 57

delayed rectifier, *see* conductance,
 delayed potassium
dendrites
 computational function, 75
 voltage-gated channels, 73
dendritic trees, 51–56
dialog box, *see* XODUS widgets, dialog box
diffusive coupling, 135, 138

EEG, *see* electroencephalogram
electroencephalogram, 154–158, 162, 164
electrotonic length, 59, 70, 244
endogenous burster, 99
enzyme, 172
EPSP, *see* postsynaptic potential,
 excitatory
equilibrium potential, 11, 30, 36, 81, 115, 220, 230
equivalent circuit, 10–12, 34, 80, 203
equivalent cylinder, 9, 62–63

field potential, 154
firing patterns, 99, 106, 108, 126
frequency encoding, 111

G protein, 173
gaits, 144, 145
gap junction, 325, 339
gates, 36
GENESIS
 cell descriptor file, 255, 266, 273
 cell reader, 255
 cell reader options, 260–262
 clocks, 222, 330
 command options, 207
 commands, 205
 data fields, 207
 data types, 217
 description, 17–20
 element, 205
 element hierarchy, 206, 209, 404
 extended objects, 236–240, 256, 283, 284
 file I/O, 374
 functions, 205, 215–218
 help, 25, 206
 interpol_structs, 305, 390
 messages, 209
 object, 205
 prototypes, 256, 266, 282, 397
 Reference Manual, 204
 script, 211
 SIMPATH, 231, 257, 263, 269
 state variable, 208
 wildcard symbol, 126, 399
 working element, 207, 238
GENESIS actions
 CHECK, 212
 CREATE, 238
 DUPLICATE, 346
 HPUT, 341
 PROCESS, 212, 240

- RESET, 212, 234, 398
RESTORE, 341
SET, 237
TABCREATE, 305
TABFILL, 307, 320
GENESIS tutorials, 4, 20–21
burster, 118
Cable, 14, 68
CPG, 133, 136
MultiCell, 232, 252
Neuron, 23, 90
Orient_tut, 253, 280
Piriform, 158
running, 22, 204
Squid, 41
traub91, 74, 122, 311
Tritonia, 132
Goldman–Hodgkin–Katz equation, 33, 314

Hebbian synapse, 251, 324
Hines method, 14
hyperpolarization, 86, 106

inactivation, 45, 102
 frequency-dependent, 104
initial conditions, 137, 144
input resistance, 55, 62, 76, 84
integrate-and-fire, 330
integration, *see* numerical integration
ion channel, 36, 171
ionic conductance, *see* conductance
IPSP, *see* postsynaptic potential, inhibitory

Kinetikit, 180–189
Kirchoff’s current law, 81, 156

lateral olfactory tract, 152
leakage conductance, *see* conductance, leakage
leakage current, 34
leakage potential, 220, 234
learning, 251
limit cycle, 134
locomotion, 132, 139, 140, 144
long term potentiation, 170, 251, 324
LTP, *see* long term potentiation

membrane
 capacitance, 11, 57, 81, 114, 218
 potential, 10
 resistance, 11, 55, 57, 81, 114, 218
 time constant, 58, 61, 82
membrane voltage, 81
Michaelis-Menten kinetics, 175
modeling
 compartmental, 8, 65–71, 244
 connectionist, 96, 195, 198
 levels of, 199
 networks, 13, 21, 131, 151, 199–201, 282, 346, 396
 structurally realistic, 196
multiprocessors, 351

negative resistance characteristic, 106
Nernst potential, 32, 317
Netkit, 396
networks, *see* modeling, networks
Neurokit, 112, 118–126, 262, 265–277
 cell editing, 126, 274, 275
 channel editing, 120, 266, 307–310
 user-variables, 266–272
 `userprefs` file, 266–272
neurokit/prototypes directory, 229, 231, 263, 266, 301, 302, 311, 320
neuron
 “Platonic”, 97
 α -motoneuron, 54

- Anisidoris* beater, 117
Aplysia L10, 99
Aplysia L11, 127
Aplysia L2, 117
Aplysia R15, 99, 105, 112
Aplysia R3, 99
 Purkinje cell, 8, 54
 pyramidal cell, 7, 54, 121, 151
 thalamic relay, 108, 276
Tritonia burster, 116, 117
 neuroscience texts, 4
 NMDA, *see* conductance, NMDA, 172
 non-physiological responses, 200
 numerical instability, 335, 336, 347
 numerical integration, 14, 332
 - Adams-Basforth methods, 333
 - backward Euler method, 14, 334
 - Crank-Nicholson method, 14, 335
 - cumulative error, 333
 - explicit methods, 14, 334
 - exponential Euler method, 14, 334, 336
 - forward Euler method, 14, 333
 - Hines method, 14, 337
 - implicit methods, 14, 243, 334
 - time step, 14, 70, 160, 222, 243
 - truncation error, 333
- object-oriented programming, 19, 212
 Ohm's law, 12, 57
 olfactory bulb, 150, 152, 162
 olfactory cortex, *see* piriform cortex
 orientation selectivity, 280
 oscillations
 - 40 Hz, 158, 162
 - phase locking, 136, 137
 - theta rhythm, 156
 - traveling wave, 142
- oscillator death, 139
 pacemaker, 99, 114
 parallel computing, 18, 113, 331, 349
 parameter search, 112, 113, 162, 331, 350, 368, 396
 passive properties, 10, 55
 PGENESIS, 349
 - pgenesis* script, 361
 - asynchronous execution, 354, 358
 - barrier, 354
 - deadlock, 354, 377
 - debugging, 362
 - file I/O, 374
 - lookahead, 367
 - namespace, 353
 - node, 352, 353
 - remote function call, 354, 357
 - script development, 352
 - synchronization, 354, 376
 - threads, 354
 - zone, 353
- phase equation model, 134, 140
 phase plane analysis, 113
 phase response curve, 138
 physiologists' convention, *see* sign conventions, physiologists'
 piriform cortex, 150
 post-hyperpolarization rebound, *see* anode break
 postsynaptic potential, 79, 86–95
 - excitatory, 89
 - inhibitory, 89
 - nonlinearity of, 87, 93, 95
- propagation delay, 247, 248, 292
 propagation velocity, 71, 292
 proprioceptors, 131
 PSP, *see* postsynaptic potential
 R_A , *see* specific, axial resistance
 R_a , *see* axial resistance
 Rallpacks benchmarks, 15

- rate constant, 37, 44, 120, 230, 311
receptor, 171, 172
 G protein coupled, 172
 ligand-gated ion channel, 172
refractory period
 absolute, 49, 249, 258
 relative, 49
rest potential, 11, 114, 219
reversal potential, *see* equilibrium potential
rheobase current, 48
 R_M , *see* specific, membrane resistance
 R_m , *see* membrane resistance
- Script Language Interpreter, 19, 211
Scripts directory, 22, 204
second messenger, 170
SI, *see* units, SI
sigmoidal, 40, 96, 104
sign conventions, 12, 35, 311
 physiologists', 35, 82, 156
signaling pathway, 169, 170, 172–175
 adenylyl cyclase, 170
 G protein activation, 170
 ligand-gated, 170
 modeling, 175
 protein kinase A activation, 170
silent inhibition, *see* synapse, silent
SLI, *see* Script Language Interpreter
space clamp, 31
space constant, 58, 61
specific
 axial resistance, 57, 218
 membrane capacitance, 57, 114, 218
 membrane resistance, 57, 82, 114,
 218
state variable, 40, 46, 102, 115, 117, 120,
 230, 303, 304
stiff equations, 335
synapse, 12, 55
dendrodendritic, 75, 326
fast, 85
inhibitory, 88
silent, 86–88, 94
slow, 88
synaptic coupling, 138, 248, 287, 400
synaptic plasticity, 13
synaptic weight, 13, 91, 163, 247, 287,
 294
- T-current, *see* conductance, low threshold calcium
temporal summation, 89, 91, 92
time constants
 equalizing, 61
 membrane, *see* membrane, time constant
 peeling of, 61
time step, *see* numerical integration, time step
transient potassium conductance, *see* conductance, A-current
Traub model, 121, 201
tridiagonal matrix, 337
Tritonia, 132
tutorial simulations, *see* GENESIS tutorials
- units, 218
 physiological, 82, 218, 311
 SI, 82, 218, 311
- V_m , *see* membrane potential
voltage clamp, 31, 41, 43
- widget, *see* XODUS widgets
- XODUS, 19, 22, 209, 382
 colorscales, 390, 393
 events, 394

-script option, 211, 223, 388, 394
XODUS widgets, 23
axis, 389
button, 24, 210
cell, 394
coredraw, 384
dialog box, 24, 228
draw, 280, 382, 385
dumbdraw, 384
form, 23, 209
geometry fields, 226
gif, 389
graph, 209, 384
label, 227
pix, 383, 387
plot, 389
shape, 389
sphere, 388
toggle, 24, 249
transformations, 385
tree, 394
var, 390
view, 393